Programming by Numbers: A Programming Method for Novices

HUGH GLASER, PIETER H. HARTEL AND PAUL W. GARRATT

Department of Electronics and Computer Science, University of Southampton, S017 1BJ Southampton, UK
Email: hg@ecs.soton.ac.uk

Students often have difficulty with the minutiae of program construction. We introduce the idea of 'Programming by Numbers', which breaks some of the programming process down into smaller steps, giving such students a way into the process of Programming in the Small. Programming by Numbers does not add intellectual difficulty to learning programming, as it does not require the student to learn additional tools or theory. In fact it can be done with pencil and paper or the normal editor, and only requires the student to remember (and understand) seven simple steps. Programming by Numbers works best with languages that offer pattern matching, such as ML, or data-directed dispatching, such as Java.

Received 24 August, 1998; revised 22 June, 2000

1. INTRODUCTION

Programming in the Small is easy for experienced programmers. First they think about how to do it (The Idea), and then they write the solution down in the language of their choice. This step from a solution in some form, to a solution in a form acceptable as input to a computer is an essential part of the expert's problem-solving skills.

Many of our students are inexperienced programmers, who often have difficulty [1]. They confuse The Idea with the program, and consequently have difficulty in constructing the program. Even if the students have been taught enough of the syntax and semantics of the language to be able to read and understand the solution when it is presented to them, they are unable to conceive of the solution. Students stare at a blank piece of paper and become frustrated.

The problem arises because there are too many issues involved. The student is unsure of where to focus, and does not have the discipline to consider only those parts of the process that are relevant at a given time, ignoring those that are not. Breaking down any process into a number of well-defined steps is the standard way of coping with such problems. We should therefore do the same with programming. A particular problem is that students may worry about the complexities of the problem before completing the simplicities; our method will require them to complete the simplicities before thinking about the complexities.

Programming by Numbers works best with a programming language that offers pattern matching, or data-directed dispatching. We have used Programming by Numbers in two different first programming courses: the first course is based on ML [2] and the second is based on Java [3]. In the ML-based course Programming by Numbers is used right from the start, because recursion is so fundamental to programming in a functional language. In the Java-based course Programming by Numbers is introduced later, when discussing recursion in connection with abstract classes, sub-classing and data-directed dispatching.

Programming by Numbers is intended to help the student get started on a problem. It is not a substitute for techniques such as step-wise refinement, structured design, or for learning about algorithms and data structures. In fact Programming by Numbers is complementary to such techniques and learning activities: it naturally leads into step-wise refinement, and it encourages the student to learn more about data structures and algorithms.

Programming by Numbers reduces the complexity of the programming task by breaking it into a number of smaller steps. Alternative approaches include plan-based programming [4], using schemata [5], or using skeletons [6]. Each of these requires the student to choose a 'template' and then to complete it in some way. This has two disadvantages.

- The students must choose the appropriate template before the problem is sufficiently well understood, or else they must change the template along the way. Templates can thus be too rigid.
- Templates generally do not help the students decide which steps must be carried out first.

Programming by Numbers is different in both respects: it provides step by step guidance, and at the same time allows flexibility in the structure of the solution to a problem. We believe that Programming by Numbers provides a better match to the creative process of programming than the alternative template-based approaches.

The underlying ideas of Programming by Numbers are not new. In fact the basic idea is so obvious that we expect many colleagues implicitly to use similar ideas. However, to our knowledge no reports exist in the literature that document a clear series of identifiable steps. Our contribution therefore is to provide careful documentation of the process, and to evaluate it by reporting on the responses of our students to the process.

Like other, similar processes (e.g. painting by numbers), Programming by Numbers has a somewhat mechanistic feel, which is not appreciated by some students. We found using Programming by Numbers most effective during lectures and laboratories, where it provides a framework for discussion. Programming by Numbers functions as an agreed and recognisable process for introducing new elements of learning and study. Each time the lecturer introduces a progressively more challenging example, there is a largely mechanistic process but also a new element. This new element appears almost automatically when we follow the steps. Once the new element has been identified, we are then able to discuss possible solutions, showing the students new data structures, algorithms, program-design techniques etc.

Finally we give our reasons for calling the method Programming by Numbers. These are:

- to elevate an important aspect of the programming process by naming the steps. It is easier to focus on a particular aspect of an intellectual process when it has been given a name;
- to order the steps. This helps to guide the student through the process;
- to facilitate memorizing the steps. People being taught complex tasks are frequently introduced to the steps by numbers (e.g. the way raw recruits are often taught gun drill).

The next section introduces Programming by Numbers using ML. Appendix A presents a number of further, graded examples, showing that a wide range of problems is covered by the method. Section 3 shows how Programming by Numbers is used with Java. An evaluation of the method on the basis of various forms of student feedback over the past three years is given in Section 4. Appendix B presents a discussion of how Programming by Numbers can be used with Z, Prolog and Pizza. Section 5 discusses the next step in the learning process and the last section presents our conclusions.

2. PROGRAMMING BY NUMBERS

Given an Idea for a function, we can provide the seven steps that a student must take to write the function.

1. Name the function.
2. Write down its type.
3. Enumerate all cases.
4. Deal with any simple case(s).
5. List the ingredients in preparation for the complex case(s).
6. Deal with the complex case(s), where some inspiration is required.
7. Think about the result.

2.1. An example

Before discussing each of the steps in more detail, we provide here as an example the construction of the `factorial` function. The student is given the assignment below. This is not the first problem a student would be asked to solve but is suitable to allow us to explain the method.

Problem: Write a function that takes a number n, and returns its factorial (i.e. $1 \times 2 \times \cdots \times n$).

1. Name the function

   ```
   factorial
   ```

   This follows directly from the statement.
2. Write down its type

   ```
   int -> int
   ```

   ML, like most other programming languages, offers two numeric types: `int` and `real`. The latter is clearly inappropriate in this case.
3. Enumerate all cases

   ```
   fun factorial 0 = refine
   |   factorial n = refine
   ```

   The identification of the cases follows directly from steps 1 and 2. For every argument type there is a standard set of cases for the initial solution attempt. For the type `int`, the usual cases are `0` and `n`, where the latter stands for non-zero. The right-hand sides have been annotated with ‘refine’, as a reminder for the refinement in the following steps.
4. Deal with any simple case(s)

   ```
   fun factorial 0 = 1
   |   factorial n = refine
   ```

   This requires a little thought, but is usually an appeal to the statement or knowledge of the problem. In this case the student has realized that 1 is the identity of multiplication. The second case has been identified as the complex case and it is still marked as ‘refine’.
5. List the ingredients

   The ‘ingredients’ available to the complex case are: the function name (`factorial`), the argument (`n`), constants of the same type as the domain and range of the function (such as `1`), and built-in functions over those types (such as `*` and `-`). From those ingredients we can construct promising expressions such as `n-1` and `factorial (n-1)`.
6. Deal with the complex case(s)

   ```
   fun factorial 0 = 1
   |   factorial n = n * factorial(n-1) ;
   ```

   The ‘difficult bit’.
7. Think about the result

   Careful examination reveals that we should be satisfied with the function defined.

The reader may wonder why we do not appeal to the theories (e.g. recursive-function theory) behind the practical method. We have experienced that it is best to avoid introducing the method from a theoretical perspective because our students are more motivated by practical issues.

2.2. A closer look

We now look at each of the steps in more detail.

2.2.1. *Name the function*

This may be a simple activity, but it is important nevertheless. Students have a tendency to underestimate the importance of choosing good names, and identifying the choice of name as an important step is useful.

2.2.2. *Write down its type*

This step takes intellectual activity because it is a formal statement of what has been informally mentioned in the question. The student is required to think about the concrete type that represents the domain of the function as indicated in the question. The example problem above indicates that the domain of the factorial function is $\mathbb{N}$. In the implementation language available, the only suitable type is `int`. The student also knows that values of type `int` lie between `minint` and `maxint`. Students can consequently be concerned that the factorial of negative numbers is not defined, and that the factorial of large numbers cannot be computed in the given range. The end result will thus be a partial function.

Writing down the type of a function exposes the student to the limitations of the function they are implementing, and the way it can be used sensibly. Students will find identifying the domain of a function difficult, but the process of Programming by Numbers helps them to focus on the question of identifying the domain. Mapping the domain to an appropriate concrete type is a problem in any programming language, and we believe that the focus on this issue provided by step 2 is helpful.

The statement of the type can be used as guidance in the remaining steps. Writing down the type also provides a method of double-checking the function when it is finally implemented. The current sub-problem of writing down the type can be solved using knowledge of the well-defined sub-language of types, thus helping the students to concentrate on one particular aspect of the solution.

2.2.3. *Enumerate all cases*

This step follows directly from step 2, and it requires the choice of appropriate names. In the case of built-in types, the base and general cases are promising:

`int`	`0`	
	non-zero	(e.g. `n`)
`bool`	`true`	
	`false`	
`string`	`""`	
	non-empty	(e.g. `s`)
`list`	`[]`	
	non-empty	(e.g. `(a::x)`)

During the early stages of the learning process we do not expect the students to create user-defined data types. Instead non-standard types are given by the lecturer. In the case of such *lecturer-defined data types*, the cases can be ascertained directly by reference to the constructors from the data-type definition. For example, it can be seen that the data type below has two patterns, one for `Empty` and one for `Push(...,...)`.

```
datatype 'a stack = Empty
                  | Push of ('a * 'a stack);
```

Where a function has more than one argument, the starting set of patterns is found by listing all combinations of the cases for the individual arguments, even though the number of cases grows exponentially with the number of arguments. The reason is that Programming by Numbers is aimed at students who do not yet have the confidence or the experience to be able to decide which cases may be combined and which may not. Having to list all combinations first and then solving the corresponding sub-problems has three advantages. Listing all cases:

- gives the student a starting point;
- gives students of all abilities the incentive to think carefully about relevant cases, and cases that can be combined;
- gives students a handle on testing their functions. A comprehensive set of test cases should at least include all combinations of the individual cases of the arguments.

Confident students see the listing of all cases as a pointless exercise. This is a disadvantage. In practice functions with more than two arguments are rare. For example out of 29 functions discussed in the first chapter on functions from the textbook we use [2, Chapter 3], 15 functions have one argument, 12 have two arguments, and only one has three arguments. These counts exclude functions that are only used as an exercise in deriving types.

2.2.4. *Deal with any simple case(s)*

Some thought is required here, but with discipline this can be minimized. The important thing is to insist on only solving the case under consideration. Concern about any other cases must be avoided. The worry about other cases is often what students find hard. It is in fact the provision of an environment where students focus on a case by case construction that motivates the Programming by Numbers method.

The simple cases are those that the student can solve with relative ease. The simple case does not have to be a base case of the recursion; we encourage the students to decide which are the simple and the complex cases.

Often the domain of the function to be written and the range of values provided by the type chosen for the implementation do not match exactly. It may then be necessary to introduce further cases. For example, the `last_element` function, which gives the last element of a list, is not defined for the empty list. This makes it necessary to introduce the case for a single-element list as the simple case.

Conversely, sometimes it turns out that simple cases can be discarded, for example when the argument concerned is not used in the function, either because it is passed directly somewhere else or because it is ignored. As experience grows, such cases can be identified in step three, but such advanced techniques should not be attempted until the basic steps are well understood.

Some students may have difficulty deciding which cases are the complex cases and which are the simple cases. In such a case it often helps to think about sample test cases for the function. For example all students would know that `factorial 0` should produce 1, thus recognizing this as a simple case. In addition they might also earmark `factorial 1` as a simple case, which while redundant is certainly correct. Most students would probably then consider `factorial n` as the complex case.

2.2.5. *List the ingredients*

Listing all elements that may be used to construct the complex case complements the information that is available about the structure of the function as a whole. For some problems, it might be sufficient to think about available built-in functions and constants. For other problems it might be necessary to postulate auxiliary functions, to reduce the difficulty of actually doing the complex case. Again an appeal to the type of the function gives guidance. For example in step 5 of our factorial problem we included `1` and `-` as ingredients, because they are clearly associated with the type `int`.

As a general rule we recommend the students consider:

- the name of the function under construction (i.e. do we need to make a recursive call?);
- the names of the function arguments, and the identifiers bound by patterns for the individual cases (i.e. how are we going to use the data available to us?);
- constants and operators over the data types of the arguments and the function result;
- in some cases auxiliary functions may be needed. The name and functionality of such auxiliary functions should be postulated. This is just an instance of step-wise refinement, which shows how Programming by Numbers leads into rather than attempts to replace tried and tested methods.

It is often useful to consider which elements are not ingredients, so as to delimit the scope of the problem. Factorial is so simple that it does not provide good examples of 'non-ingredients'. We give some examples and a discussion in the next section.

2.2.6. *Deal with the complex case(s)*

Solving the complex case(s) can be difficult, but even here it is possible to apply some heuristics. Either there is a relatively simple answer, or we need to divide the solution (i.e. the right-hand side) into some sub-problems.

In turn, the identification of sub-problems can be hard, but the important thing is that a sub-problem should be smaller in some sense than the original problem. In the case of a function over a list, for example, we expect the solution to contain some computation on the tail of the list. In the case of integers, as in factorial above, we expect the solution to contain some computation on a smaller number (one less usually), such as `factorial (n-1)` above.

Having identified and solved a suitable sub-problem, the next step is to place the solution in an expression such that the equality constraint is satisfied. To achieve this for factorial, we need to use it as the right operand in `n * factorial (n-1)`.

For a large set of problems, the only step that a student who is applying the method with discipline will find challenging is the current step. In fact this step is the nub of any algorithm, and so the student is required to concentrate on the important part of the function without being distracted by all the book-keeping. Programming by Numbers has successfully broken a problem down into smaller sub-problems. One should now appeal to the student's understanding of the problem, knowledge of algorithms and data structures, and skills in applying step-wise refinement.

2.2.7. *Think about the result*

Finally we need to review the function written to satisfy ourselves that it is what we want. This step is sometimes null, as with factorial. Normally we recommend our students to consider the following issues:

- Functions are static objects, which can sometimes be usefully simplified. The most common simplification is to realize that some of the special cases are not in fact special, and are also covered by more complex cases. Making such a simplification leads to a better insight into the original problem.
- Checks should be made, verifying that the type of the function is the same as, or a generalization of, the type from step 2. Finding type generalization often goes hand in hand with removing redundant cases.
- It is useful at this stage to write some test cases for the function to exercise the simple and complex cases.
- Frequently the simplifications, checks and tests performed at this stage lead to new insights which cause the student to reconsider earlier steps, making Programming by Numbers an iterative process.

3. USING JAVA

Programming by Numbers emphasizes case analysis, which works well in programming languages like ML. To show how Programming by Numbers works in object-oriented languages like Java consider the following problem.

Problem: Write a method to sum the leaves of a binary tree, where the tree has been defined (by the lecturer) as an abstract class `Tree` with sub-classes `Leaf` and `Branch`.

1. Lecturer-defined data type

```
public abstract class Tree {
    refine methods
}
public class Leaf extends Tree {
   public Leaf() {
   }
    refine methods
}
public class Branch extends Tree {
   private Tree left, right;
   private int data;
   public Branch(Tree left, int data
                 Tree right) {
      this.left = left;
      this.data = data;
      this.right = right;
   }
    refine methods
}
```

The lecturer-defined data type describes the data structure, complete with fields and initializers. The student is asked to provide the definition of the appropriate method(s). Here is how the process would be executed.

1. Name the function(method)

   ```
   sum
   ```

   A good name for the method is `sum`.
2. Write down its type

   ```
   public abstract class Tree {
      abstract int sum();
   }
   ```

   The student is unlikely to define `sum` as a function with a `Tree` argument because it is more befitting of a Java program to use a method. The method does not need arguments. Given that the values contained in our tree are integers, the return type for the method is `int`.
3. Enumerate all cases

   ```
   public class Leaf extends Tree {
      // initialiser omitted
      public int sum() {
          refine body
      }
   }
   public class Branch extends Tree {
      // initialiser and fields omitted
      public int sum() {
          refine body
      }
   }
   ```

   Where we relied on pattern matching in the ML example, we now use Java's data directed dispatching to identify the two relevant cases: one to handle a `Leaf` and one to deal with a `Branch`. It suffices to write each case in the appropriate class.
4. Deal with any simple case(s)

   ```
   public class Leaf extends Tree {
      // initialiser omitted
      public int sum() {
         return 0 ;
      }
   }
   ```

   Most students would identify the `Leaf` as the simple case, because all that needs to be done here is to return 0. Students who choose `Branch` as the simple case will probably find it difficult to complete this step and are likely to revisit their earlier, incorrect decision.
5. List the ingredients

   `sum`, `+`, `left`, `data`, and `right`

   It is necessary to think carefully about the list of ingredients for the complex case. Using the listed ingredients, the student may form such expressions as `left.sum()`, and `right.sum()` which would contribute to finding a solution at step 6.
   We encourage students to explore further alternatives, for example `left.left`, and ask them whether such expressions make sense under all circumstances, a test which the expression `left.left` does not pass.
6. Deal with the complex case(s)

   ```
   public class Branch extends Tree {
      // initialiser and fields omitted
      public int sum() {
         return left.sum() + data + right.sum();
      }
   }
   ```

   Having given the list of ingredients sufficient thought, most students would not find it difficult to complete the complex case. The complete solution consisting of three classes is then as shown below.
7. Think about the result

   ```
   public abstract class Tree {
      abstract int sum();
   }
   public class Leaf extends Tree {
      public Leaf() {
      }
      public int sum() {
         return 0 ;
      }
   }
   public class Branch extends Tree {
      private Tree left, right;
      private int data;
      public Branch(Tree left, int data,
                    Tree right) {
         this.left = left;
         this.data = data;
         this.right = right;
      }
      public int sum() {
         return left.sum() + data + right.sum();
      }
   }
   ```

We have taken the liberty here of varying the method slightly by using step 7 to assemble the elements created in steps 4 and 6.

After giving some thought to the solution we cannot come up with improvements. However we can think of various interesting test cases for the `sum()` method. We will not show them here.

It is interesting to compare our solution above to what one would encounter in a typical text book. Deitel and Deitel [7, Section 17.7] use a single class `TreeNode`, with three fields, to represent a binary search tree:

```
class TreeNode {
   TreeNode left;
   int data;
   TreeNode right;
    methods
}
```

A leaf node is represented by an instance of `TreeNode` with the fields `left` and `right` set to `null`. The approach of Deitel and Deitel has two problems:

- The implementation of the `TreeNode` is exposed and is thus not an abstract data type.
- It is not possible to distinguish between accessing an uninitialized reference (usually a programming error) and an empty (sub-) tree.

Deitel and Deitel's solution is probably easier for the students to understand, but with the help of Programming by Numbers we believe the students will appreciate our solution as the better one.

4. EVALUATION

Programming by Numbers has been used for the last eight years in two different first courses on Programming Principles for Computer Science students at the University of Southampton [8]. This year the course is based on Java, using the text book by Arnow and Weiss [3]; previously the course was based on standard ML, using Ullman [2].

Southampton attracts a growing number of Computer Science students, presently numbering about 160 per year. The Programming principles course is assessed on the basis of 50% examination and 50% course work. Fewer than 10% of the students who complete the year are required to resit the examination in September.

To evaluate Programming by Numbers, ideally we would design an experiment with two equivalent, randomly chosen groups of students. One group would be taught using Programming by Numbers and the control group would receive the same tuition without using Programming by Numbers. However, the costs of such an experiment are prohibitive, not least because of the need to isolate the two groups of students from each other. In addition there are serious ethical issues of large-scale experimentation on such an important part of the students' degree. Consequently we have attempted to evaluate the method by canvassing the opinions of the students. We have done so using two different methods of evaluation: an observational investigation [9] in 1999/2000 and exam-based surveys in 1997/1998 and 1998/1999.

4.1. Observational investigation

To explore the utility of Programming by Numbers we conducted a formal test where we observed students attempting to solve a programming problem using Programming by Numbers and Java. Eight students (5% of the class) were chosen at random from the entire cohort. An observation lasted at most 30 minutes; some students completed their task in less time. The students had all been new to Java at the start of the course, and some had been new to computer programming. The students were all studying Computer Science but from the perspectives of different degree courses. Some were more mathematically oriented while some were more oriented towards electronics. The students were presented with Programming by Numbers in a single lecture which explained the method and finished with an example: constructing a Java method to sum the elements of a list.

For the observed experiments the students had to construct the Java method to sum the leaves of a binary tree as in Section 3, using Programming by Numbers. During the exercise an observer noted the actions of the student, noted the student's use of Programming by Numbers and examined the resulting program. The observer behaved according to the formal guidelines defined in Nielsen [9]. The rigorous conduct of such an exercise requires the observer to ensure the subject understands the problem and the role of the method but then refrains from intervening while the subject uses the method. After the observation a less formal session allows further discussion and clarification of events.

The formal observations revealed that all the students but one used Programming by Numbers in the recommended way. All students who used Programming by Numbers produced acceptable Java methods and appeared to have understood and explored abstract classes and data directed dispatching by the end of the exercise. The less experienced students used Programming by Numbers in conjunction with referring back to simpler examples of Java programs. The more experienced students referred back to example Java programs less and took less time to complete the exercise. During the final stages of each student's session the observer noted that generally the students appeared pleasantly surprised that the program they had written seemed to work. One student said he was 'shocked' at the successful outcome of his Programming by Numbers. The method had led the students to the correct solution of a non-trivial programming task in easy stages.

During the observational investigation a case arose which provided a valuable control situation for our experiment. One student when presented with the programming task and Programming by Numbers attempted the task but refrained from using the method, preferring to attempt the task in his own way. His reasons were unclear to the observer who thought they may have stemmed from over-confidence.

The student solved the programming problem to his own apparent satisfaction but the Java method he wrote was an entirely inappropriate solution to the problem. Also the student failed to learn the programming lessons implicit in the programming task and so derived little benefit compared to the students who did use Programming by Numbers.

At the close of each observed session the observer had an informal discussion with the student to clarify the outcome and gauge the student's reaction. Also at the end of the whole observational investigation the students gathered in an informal panel to discuss the purpose, results and conclusions of the investigation. The students praised Programming by Numbers and reported finding it very useful in teaching them aspects of Java. They commented that Programming by Numbers reflects the problem-solving method they use intuitively and it is a useful clarification and formalization of that method. The students agreed with our description of Programming by Numbers as limited in its scope. They said they would welcome a replication and an extension of Programming by Numbers to cover other aspects of program design and development. Some were optimistic that Programming by Numbers would ease their difficulties in comprehending and learning large-scale programming as it had helped them during the observational investigation.

Finally, the students suggested an addition to step 4, which has now been incorporated in the method. The addition consists of writing down some test cases for the function/method being developed, to help decide which are the simple and the complex cases, and to provide further insights while dealing with the cases.

4.2. Exam-based surveys

We embedded a question on Programming by Numbers in the 1997/1998 and 1998/1999 examinations, thereby ensuring that this captive audience were all encouraged to give their views. The two editions of the course were the same; both were based on standard ML. In 1997/1998 we asked the following question:

> (i) Give the different steps of Programming by Numbers (5%).
>
> (ii) Write a short essay (no more than one page) to explain and defend one of the following points of view (20%):
>
> (a) Programming by Numbers helps you get started with a programming problem;
>
> (b) Programming by Numbers does not help you get started with a programming problem.
>
> Marks will be awarded only for the effective defence of the chosen point of view.

The 1997/1998 question was answered by 28 out of 104 students; 27 of those argued case (a) and one student argued case (b). We attach no significance to these numbers because they clearly reflect the student's inclination to please their lecturers. Instead we scrutinized the exam scripts to identify positive and negative remarks about the method (below).

Given the overwhelmingly positive reaction to our 1997/1998 Programming by Numbers question we decided to ask for the disadvantages in 1998/1999:

> (i) Give the different steps of Programming by Numbers (5%).
>
> (ii) Discuss with the aid of examples the shortcomings and disadvantages of Programming by Numbers. Your answer should be concise – one page maximum (20%).

This resulted in 44 out of 127 students giving their views in 1998/1999.

4.2.1. Positive reactions in 1997/1998

Students commented that the process as a whole helped them to

- create manageable sub-tasks,
- give structure to the programming activity,
- experience a gradual progression from easy to difficult,
- put syntax and semantics into practice, and
- have 'peace of mind'. One student wrote 'you don't have to look at a blank piece of paper for ages'. Another student wrote that 'even if you can't solve a problem you can still write some code'.

These comments indicate that Programming by Numbers indeed helps to avoid the frustration of staring at a blank piece of paper.

One student gave us a new insight when he/she wrote that programs are split into separate entities (functions) so it is natural to also split functions into separate entities.

Students found having to write the type of the function helpful in various ways. They said that

- it helps you understand the problem,
- it helps you write code that someone else is able to read more easily, and
- the practice of writing down the types as a separate step can be transferred to other languages.

Enumerating all the cases was found useful 'because you are less likely to forget special cases'.

By far the most frequent comment was that doing the simple case(s) helps doing the complex case(s). Some students noted that doing the simple case(s) first also gave them the opportunity to think about termination of a recursive function.

The final step was generally quoted as useful to eliminate redundant cases, but also as an opportunity to rethink the whole development process.

4.2.2. Negative reactions in 1997/1998

One student, whilst arguing case (a) noted that it may take longer writing functions using Programming by Numbers, and he/she also wrote 'why would it exist if it was not useful?'.

The identification of the cases is meant to give students a handle on the tests to be applied during white-box testing. However we found that because of this emphasis on white-box testing the students are less likely to also apply black-box testing.

In a sense the most interesting reaction is that of the student who chose to argue case (b) of the exam question. The point made was basically that Programming by Numbers does not scale up. It does not help to write a large program. Of course the student is absolutely right, and in retrospect we should have brought this point out during the course.

4.2.3. Negative reactions in 1998/1999

About half the students noted that Programming by Numbers can be time-consuming because of the need to write down all those cases. We would argue that comprehensive testing requires writing down all the cases anyway.

About 40% of the students noted that Programming by Numbers has a limited scope because solving the difficult cases is sometimes too difficult, and it requires what students call Programming by Numbers within Programming by Numbers (step-wise refinement).

Some 30% claimed that Programming by Numbers can only be used with functional programming because it relies on pattern matching.

About 25% of the students were concerned about Programming by Numbers being restrictive and inhibiting creativity because 'people do not think in a prescribed order'. A majority of these students were especially concerned with the fact that Programming by Numbers does not necessarily lead to the most efficient solution: it is no substitute for knowledge of algorithms and data structures.

Finally three students noted that Programming by Numbers is so obvious and natural that it is not worth learning it!

5. THE NEXT STEP IN THE LEARNING PROCESS

In our standard ML-based Programming Principles course, the early assignments are relatively simple, like the factorial problem. We follow this up in the second phase of the course using assignments that allow the students to develop some of the standard higher order functions, and to use 'student'-defined data types (as opposed to 'lecturer'-defined data types). The purpose of this second stage is threefold:

- to provide further opportunity to practice programming in the small;
- to allow students to discover the utility of standard idioms as exemplified by `map` (See Appendix A.9) and `filter`;
- to practise abstraction over functions (yielding higher-order functions) and abstraction over types (yielding polymorphic functions).

The third and last phase of the Programming Principles course revisits the above, but uses C instead of ML [10]. The students feel confident that they understand the algorithms and data structures involved; the new challenge is to express these using a different, lower-level, programming language.

Programming by Numbers helps students to become *advanced beginners* [1]. Once the students are comfortable with writing single functions over familiar domains they are presented with more challenging problems. These would typically require a level of decomposition into sub-problems that can be solved directly using Programming by Numbers. It is at this stage in the learning process, when progressing from advanced beginner to *competence*, that classical problem-solving methods such as Jackson Structured Programming [11] would be appropriate for use. Such methods support the student in analysing the problem, which in turn helps the student to pick out the elements that can be solved using Programming by Numbers. It is our experience that the classical methods require the student to be able to manipulate the elementary programming constructs with a greater degree of skill than students possess. Programming by Numbers provides a method to fill this gap.

5.1. Java

In the Java-based Programming Principles course Programming by Numbers has a different role than in the standard ML-based course. The text book used [3] follows a classical approach towards imperative programming, emphasizing loops and arrays. Abstract classes, data-directed dispatching and also recursion play a minor role. However, we believe these are fundamental and important concepts, the discussion of which is continued in the two following courses: Advanced Programming and Algorithms and Data Structures.

6. CONCLUSIONS

Based on our experience of teaching programming, where we observe that students have difficulty with recursion and case analysis, we set out to develop a method specifically to help students with these problems.

Programming by Numbers is a method that gives the student a series of well-defined steps to provide the discipline for creating the smallest components of functions. The method identifies separate activities and orders them such that the student is able to progress from an empty sheet of paper to a worked solution by following the steps. The student is encouraged to direct creativity at the sub-problems. The student does not have to worry about everything at the same time.

We are aware that our insistence on following the steps, even to the point where many different cases are generated, might hinder students' creativity. Our response is that Programming by Numbers is to be used as a guide for class-room discussions. Our work provides a vehicle that the lecturer can use to discuss important issues such as 'how many cases shall we consider and why?'. We have found this question to be extremely useful in our lectures over many years.

Student feedback on the use of the method is largely positive. We gained some new insights from the students' comments, which we interpret as an indication that the students do appreciate the help provided by the method.

The method naturally leads students to think about programming idioms once they progress beyond a certain point.

The method was developed with functional programming in mind but it can be applied to any language that supports a form of case analysis. We have shown how to use Programming by Numbers with Java. The appendices show how to use Programming by Numbers with the specification language Z, the logic language Prolog, and the Java super-set Pizza.

ACKNOWLEDGEMENTS

The contributions of eight generations of students are gratefully acknowledged, in particular the students who were observed in 1999. We thank Michael Butler, Kathryn Glaser, Andy Gravell, Hugh Davis, Jon Hallett, Benoît Lanaspre, Denis Nicole, and the four anonymous referees for their help.

REFERENCES

[1] Winslow, L. E. (1996) Programming pedagogy—a psychological overview. *ACM SIGCSE Bull.*, **28**, 17–22, 25.

[2] Ullman, J. D. (1994) *Elements of ML Programming*. Prentice Hall, Englewood Cliffs, NJ.

[3] Arnow, D. and Weiss, G. (1999) *Introduction to Programming Using Java: An Object-Oriented Approach (Java 2 Update)*. Addison Wesley Longman Higher Education, Reading, MA.

[4] Hein, J. L. (1993) A declarative laboratory approach for discrete structures, logic and computability. *ACM SIGCSE Bull.*, **25**, 19–24.

[5] Bieliková, M. and Návrat, P. (1998) Use of program schemata in Lisp programming. *Informatica*, **9**, 5–20.

[6] Sterling, L. and Kirschenbaum, M. (1993) Applying techniques to skeletons. In J. M. Jacquet (ed.), *Constructing Logic Programs*, pp. 127–140. John Wiley & Sons, Chichester, UK.

[7] Deitel, H. M. and Deitel, P. J. (1998) *Java—How to Program* (2nd edn). Prentice Hall Int. Inc., Upper Saddle River, NJ.

[8] Glaser, H. and Sivess, V. (1993) Un language fonctionnel pour le cours d'initiation à la programmation. *Spécif*, **93**, 14–25.

[9] Nielsen, J. (1994) *Usability Engineering*. Academic Press, London.

[10] Hartel, P. H. and Muller, H. L. (1997) *Functional C*. Addison Wesley Longman, Harlow, UK.

[11] Ingevaldsson, L. (1979) *JSP—A Practical Method of Program Design*. Jackson Systems Int. Ltd, London.

[12] Hudak, P., Peyton Jones, S. L. and Wadler, P. L. (eds) (1992) Report on the programming language Haskell—A non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, **27**, R1–R162.

[13] Turner, D. A. (1985) Miranda: A non-strict functional language with polymorphic types. In Jouannaud, J.-P. (ed.), *2nd Intl Conf. Functional Progr. Lang. Comp. Architecture*, Nancy, France. *Lecture Notes in Computer Science*, **201**, 1–16.

[14] Elenbogen, B. C. and O'Kennon, M. R. (1998) Teaching recursion using fractals in Prolog. In H. L. Dersham (ed.), *19th Int. Conf. Computer Science Education*, Atlanta, GA, pp. 263–266. *ACM SIGCSE Bull.*, **20**.

[15] Böszörmónyi, L. (1998) Why Java is not my favorite first-course language. *Software—Concepts and Tools*, **19**, 141–145.

[16] Hosch, F. (1996) Java as a first language: an evaluation. *ACM SIGCSE Bull.*, **28**, 45–50.

[17] Spivey, J. M. (1989) *The Z Notation*. Prentice Hall, Englewood Cliffs, NJ.

[18] Gravell, A. M. and Henderson, P. (1996) Executing formal specifications need not be harmful. *Software Eng. J.*, **11**, 104–110.

[19] Stepney, S. (1993) *High Integrity Compilation: A Case Study*. Prentice Hall, Hemel Hempstead, UK.

[20] Jia, X. (1995) *ZTC: A Type Checker for Z—User's Guide*. Dept. of Comp. and Inf. Sci, DePaul Univ., Chicago, Illinois. `ftp.comlab.ox.ac.uk/ pub/ Zforum/ ZTC-1.3/ guide.ps.Z`.

[21] Sterling, L. and Shapiro, E. (1994) *The Art of Prolog*. MIT Press, Cambridge, MA.

[22] PDC (1998) *Visual Prolog 5.0 Reference Manual*. Prolog Development Center A/S, Copenhagen, Denmark. `www.visual-prolog.com`.

[23] Somogyi, Z., Henderson, F. and Conway, T. (1996) The execution algorithm of Mercury: an efficient purely declarative logic programming language. *J. Logic Programming*, **29**, 17–64.

APPENDIX A. GRADED EXAMPLES

We begin each example with a formulation of the problem, as one might find it in a typical course assignment. We then present the seven steps and comment on some of the salient features of the method and the example.

After the initial, simple functions, the rest of the examples are of recursive functions, because that is the only class of 'interesting' functions that can be discussed during the first few weeks of a first programming course.

The example problems have been chosen mainly because they appear to pose problems for the method.

A.1. A logical operation

Problem: Write a function to provide the logical negation of its argument.

1. Name the function
```
not
```
2. Write down its type
```
bool -> bool
```
3. Enumerate all cases
```
fun not true  = refine
|   not false = refine
```
4. Deal with any simple case(s)
```
fun not true  = false
|   not false = true ;
```
7. Think about the result

This problem does not have complex cases, so steps 5 and 6 can be skipped. We do insist that students explicitly make step 7 even though, as in this case, it does not give rise to new insights.

A.2. An arithmetic operation

Problem: Write a function to provide the arithmetic negation of its argument.

1. Name the function
   `negative_of`
2. Write down its type
   `int -> int`
3. Enumerate all cases
   ```
   fun negative_of 0 = refine
   |   negative_of n = refine
   ```
4. Deal with any simple case(s)
   ```
   fun negative_of 0 = 0
   |   negative_of n = refine
   ```
5. List the ingredients
   `negative_of` and `n`
6. Deal with the complex case(s)
   ```
   fun negative_of 0 = 0
   |   negative_of n = ~n ;
   ```
7. Think about the result
   ```
   fun negative_of n = ~n ;
   ```

When thinking about the function, the student realizes that the special case is covered by the complex case, because `~0=0` for the integers. (Unary minus in ML is `~`.) This example shows that the student who does not think of such an identity will still be able to produce a working function. We believe that it is important that the method should work for students of all abilities.

A.3. A list function

Problem: Write a function to return a string that has a number of copies (first argument) of its (second) argument, a string.

1. Name the function
   `copies`
2. Write down its type
   `int -> string -> string`
3. Enumerate all cases
   ```
   fun copies 0 ""  = refine
   |   copies n ""  = refine
   |   copies 0 str = refine
   |   copies n str = refine
   ```
4. Deal with any simple case(s)
   ```
   fun copies 0 ""  = ""
   |   copies n ""  = ""
   |   copies 0 str = ""
   |   copies n str = refine
   ```
5. List the ingredients
   `copies`, `n`, and `str`
6. Deal with the complex case(s)
   ```
   fun copies 0 ""  = ""
   |   copies n ""  = ""
   |   copies 0 str = ""
   |   copies n str =
           str ^ (copies (n-1) str) ;
   ```
7. Think about the result
   ```
   fun copies 0 str = ""
   |   copies n str =
           str ^ (copies (n-1) str) ;
   ```

Here ^ is string concatenation.

For this function, we see that in general for two-argument functions we must enumerate all possible combinations of arguments. It is better, as here, to enumerate all the argument pairs and then remove those redundant cases we wish in step 7, rather than risk getting it wrong by removing them earlier. This is in contrast to a common student difficulty where the attempt to 'optimize' during the construction process causes serious problems.

A.4. A list function with a problem

Problem: Write a function to return the last element of a list.

1. Name the function
   `last_element`
2. Write down its type
   `'a list -> 'a`
3. Enumerate all cases
   ```
   fun last_element [] = refine
   |   last_element (a::x) = refine
   ```
4. Deal with any simple case(s)
   ```
   fun last_element [] = problem
   |   last_element (a::[]) = a
   |   last_element (a::x) = refine
   ```
5. List the ingredients
   `last_element`, `a`, and `x`
6. Deal with the complex case(s)
   ```
   fun last_element (a::[]) = a
   |   last_element (a::x) = last_element x ;
   ```
7. Think about the result

We know that the domain of the `last_element` function is all non-empty lists. However, the only reasonable concrete type for the implementation is the standard list type, as indicated in the problem. The concrete type does not match exactly the domain of the function. Hence a new case must be introduced to deal with the singleton list. The student who does realize that the mismatch exists, will introduce a case at step 3. Those that do not realize that the mismatch exists will discover the problem at step 4 as above, when trying to think of a value to be returned when the argument is the empty list.

We could raise an exception, but at this stage we will assume the function will be used correctly, and rely on the system to raise an exception if it is not. This is achieved simply by omitting the case of the empty list from the program. The exercise has, however, ensured that we are aware of the potential danger when using this function. The method has almost insisted that the student notices the function is partial. The student is encouraged to focus on the reasons and is isolated from the complexities of the whole function.

The student might also try the alternative approach shown below. This solution acknowledges the fact that pattern matching has limitations. Sometimes conditionals are needed to deal with cases that pattern matching has difficulty with. We would argue that this is not a limitation of Programming by Numbers, but it is more due to the lack of guards in ML. Examples such as this one would be dealt with in a more natural way in Haskell [12] and Miranda [13] which do offer guards in addition to pattern matching. We give further examples of using conditionals in subsequent sections.

4. Deal with any simple case(s)
```
fun last_element [] = problem
|   last_element (a::x) =
        if null x
        then a
        else  refine
```
5. List the ingredients
`last_element`, `a`, and `x`
6. Deal with the complex case(s)
```
fun last_element (a::x) =
        if null x
        then a
        else last_element x ;
```
7. Think about the result

A.5. A function of two lists

Problem: Write a function to add the elements of two lists together, for example the result of adding the lists `[1,2,3]` and `[4,5,6]` should be the list `[5,7,9]`.

1. Name the function
`add_lists`
2. Write down its type
`int list -> int list -> int list`
3. Enumerate all cases
```
fun add_lists [] [] = refine
|   add_lists [] (a::x) = refine
|   add_lists (a::x) [] = refine
|   add_lists (a1::x1) (a2::x2) = refine
```
4. Deal with any simple case(s)
```
fun add_lists [] [] = []
|   add_lists [] (a::x) = problem
|   add_lists (a::x) [] = problem
|   add_lists (a1::x1) (a2::x2) = refine
```
5. List the ingredients
`add_lists`, `a1`, `x1`, `a2`, `x2`
6. Deal with the complex case(s)
```
fun add_lists [] [] = []
|   add_lists (a1::x1) (a2::x2) =
        (a1+a2) :: (add_lists x1 x2) ;
```
7. Think about the result

At step 4 the student realizes that the problem statement is unclear, because it does not specify what to do with lists of unequal length. It is good that this difficulty does not arise at the same time as when we are thinking about the complex step. In the example we leave `add_lists` as a partial function.

A.6. A three-argument function

Problem: Write a function to return a list that has an element (first argument) inserted into the *n*th position (second argument) of a list (third argument).

1. Name the function
`insert`
2. Write down its type
`'a -> int -> 'a list -> 'a list`
3. Enumerate all cases
```
fun insert item 0 [] = refine
|   insert item 0 (a::x) = refine
|   insert item n [] = refine
|   insert item n (a::x) = refine
```
4. Deal with any simple case(s)
```
fun insert item 0 [] = item :: []
|   insert item 0 (a::x) = item :: (a::x)
|   insert item n [] = problem
|   insert item n (a::x) = refine
```
5. List the ingredients
`insert`, `item`, `n`, `a`, `x`
6. Deal with the complex case(s)
```
fun insert item 0 [] = item :: []
|   insert item 0 (a::x) = item :: (a::x)
|   insert item n (a::x) =
        a :: (insert item (n-1) x) ;
```
7. Think about the result
```
fun insert item 0 lst = item :: lst
|   insert item n (a::x) =
        a :: (insert item (n-1) x) ;
```

In the case of a three-argument function, there is a potentially large number of cases. Fortunately, in writing the type of the first argument as polymorphic, the student is encouraged to realize that there is no need (or even a possibility) of identifying different cases for that argument.

A.7. More challenging—sorting

Problem: Write a function to return the sorted version of its argument list (we will consider integers in this example).

For such a function we really need an Idea. In this case the Idea is that we take the first element and then divide the list into elements which are `less_than` it (putting those at the front), and those that are `not_less_than` (putting those afterwards).

1. Name the function
`sort`
2. Write down its type
`int list -> int list`
3. Enumerate all cases
```
fun sort [] = refine
|   sort (a::x) = refine
```
4. Deal with any simple case(s)
```
fun sort [] = []
|   sort (a::x) = refine
```
5. List the ingredients
`sort`, `a`, `x`, and
postulate `less_than` and `not_less_than`
6. Deal with the complex case(s)
```
fun sort [] = []
|   sort (a::x) = sort (less_than a x)
        @ [a]
        @ sort (not_less_than a x) ;
```
7. Think about the result

Here `@` is the concatenation operator on lists.

Now the new functions must be defined. We will only define one, as the other is similar.

1. Name the function
`less_than`
2. Write down its type
`int -> int list -> int list`

3. Enumerate all cases
   ```
   fun less_than 0 [] = refine
   |   less_than 0 (a::x) = refine
   |   less_than pivot [] = refine
   |   less_than pivot (a::x) = refine
   ```
4. Deal with any simple case(s)
   ```
   fun less_than 0 [] = []
   |   less_than 0 (a::x) = refine
   |   less_than pivot [] = []
   |   less_than pivot (a::x) = refine
   ```
5. List the ingredients
   `less_than`, `0`, `pivot`, `a`, `x`
6. Deal with the complex case(s)
   ```
   fun less_than 0 [] = []
   |   less_than 0 (a::x) =
          if a < 0
          then a :: (less_than 0 x)
          else (less_than 0 x)
   |   less_than pivot [] = []
   |   less_than pivot (a::x) =
          if a < pivot
          then a :: (less_than pivot x)
          else (less_than pivot x) ;
   ```
7. Think about the result
   ```
   fun less_than pivot [] = []
   |   less_than pivot (a::x) =
          if a < pivot
          then a :: (less_than pivot x)
          else (less_than pivot x) ;
   ```

A.8. User-defined types

Non-recursive ‘lecturer’-defined types are quite straightforward. Recursive types can be dealt with as before.

Problem: Write a function to sum the leaves of a tree, where the tree has been defined by the lecturer as follows.

0. Lecturer-defined data type
   ```
   datatype tree = Leaf
                 | Branch of (tree * int * tree)
   ```
1. Name the function
   `sum`
2. Write down its type
   `tree -> int`
3. Enumerate all cases
   ```
   fun sum (Leaf) = refine
   |   sum (Branch(left,data,right)) = refine
   ```
4. Deal with any simple case(s)
   ```
   fun sum (Leaf) = 0
   |   sum (Branch(left,data,right)) = refine
   ```
5. List the ingredients
   `sum`, `+`, `left`, `data`, and `right`
6. Deal with the complex case(s)
   ```
   fun sum (Leaf) = 0
   |   sum (Branch(left,data,right)) =
       (sum left) + data + (sum right) ;
   ```
7. Think about the result

A.9. Mapping a function over a list

Problem: Given a function `inc : int -> int`, write a new function to increment each element of an integer list.

1. Name the function
   `inc_list`
2. Write down its type
   `int list -> int list`
3. Enumerate all cases
   ```
   fun inc_list [] = refine
   |   inc_list (a::x) = refine
   ```
4. Deal with any simple case(s)
   ```
   fun inc_list [] = []
   |   inc_list (a::x) = refine
   ```
5. List the ingredients
   `inc`, `inc_list`, `a`, `x`
6. Deal with the complex case(s)
   ```
   fun inc_list [] = []
   |   inc_list (a::x) = inc a :: inc_list x ;
   ```
7. Think about the result

The less experienced programmer will go through all the seven steps, and produce the correct function above. A more experienced programmer would recognize an idiom at some stage, and if necessary retrace some of the steps to come up with this solution:

1. Name the function
   `inc_list`
2. Write down its type
   `int list -> int list`
3. Enumerate all cases
   ```
   fun inc_list x =  refine
   ```
4. Deal with any simple case(s)
   ```
   fun inc_list x = map inc x;
   ```
7. Think about the result
   ```
   val inc_list = map inc;
   ```

Programming by Numbers is designed to help students get started. This example shows that the method helps to develop the skills required to recognize and use standard idioms.

A.10. Fibonacci

Problem: Write a function to compute the nth element from the Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, \ldots$, in which each number is the sum of the previous two.

1. Name the function
   `fib`
2. Write down its type
   `int->int`
3. Enumerate all cases
   ```
   fun fib 0 = refine
   |   fib n = refine
   ```
4. Deal with any simple case(s)
   ```
   fun fib 0 = 0
   |   fib 1 = 1
   |   fib n = refine
   ```
5. List the ingredients
   `fib`, `n`
6. Deal with the complex case(s)
   ```
   fun fib 0 = 0
   |   fib 1 = 1
   |   fib n = fib (n-1) + fib (n-2);
   ```
7. Think about the result

At step 4 some students realize that in order to compute a number in the Fibonacci sequence, the previous two must be given. Therefore, there must be at least two simple cases, one for 0 and one for 1. Other students will discover that there is a problem when they think about the function, and others may only realize the problem when they test a solution that has insufficient base cases.

Some students will realize that the resulting function has an exponential complexity. This problem can only be solved by using a better algorithm, which computes the next Fibonacci number given the previous two. This problem can be solved by using an auxiliary function `twofib` that returns the current as well as the previous Fibonacci number. Once that has been decided, there is no difficulty in applying Programming by Numbers again to the new programming problem. Here we show the auxiliary function `twofib`.

1. Name the function
   `twofib`
2. Write down its type
   `int->(int*int)`
3. Enumerate all cases
   ```
   fun twofib 0 = refine
   |   twofib n = refine
   ```
4. Deal with any simple case(s)
   ```
   fun twofib 0 = (0,1)
   |   twofib n = refine
   ```
5. List the ingredients
   `twofib, n`
6. Deal with the complex case(s)
   ```
   fun twofib 0 = (0,1)
   |   twofib n = let val (a,b) = twofib (n-1)
                  in (b,a+b) end;
   ```
7. Think about the result

A.11. Ascending sequence of integers

Problem: Write a function to compute an ascending sequence of integers n, $n+1$, $\ldots, m$. The statement of the problem suggests that the cases to be considered should include numbers at the *beginning* of the range as well as the *end* of the range. Here we encounter a limitation, not so much of Programming by Numbers, but more of ML, which does not allow non-linear patterns (Miranda for example does allow such patterns). To cope with cases involving comparisons we need to use a conditional.

1. Name the function
   `range`
2. Write down its type
   `int -> int -> int list`
3. Enumerate all cases
   ```
   fun range n m =
           refine, where n < m
           refine, where n = m
           refine, where n > m
   ```
4. Deal with any simple case(s)
   ```
   fun range n m =
           refine, where n < m
           [n]   where n = m
           []    where n > m
   ```
5. List the ingredients
   `range, n m`
6. Deal with the complex case(s)
   ```
   fun range n m =
           n :: range (n+1) m where n < m
           [n]   where n = m
           []    where n > m
   ```
7. Think about the result
   ```
   fun range n m =
           if n <= m then n :: range (n+1) m
           else []
   ```

APPENDIX B. USING Z, PROLOG AND PIZZA

We now show how Programming by Numbers can be applied with the specification language Z, the logic-programming language Prolog, and the object-oriented-programming language Pizza.

None of these three languages are mainstream first programming languages. However, Z and Prolog [14] are used in the courses on discrete mathematics that often run in parallel with the first programming course. We include Pizza in our selection of languages because we feel that it would be an interesting alternative to Java as a first language [15, 16]. No reports are available as yet on the use of Pizza as the first programming language but we have heard reports from various institutions contemplating this step.

B.1. Z

The specification language Z [17] is used in many institutions to teach formal methods to first-year students. Full Z is not normally used to introduce programming but various executable subsets of Z have been created for the purpose of animating specifications [18]. Here we consider how Programming by Numbers might be adapted to teach programming in a 'functional' sub-set of Z. This style of specification is not mainstream Z. However, there are reports in the literature where this particular style of specification is used extensively [19].

Problem: Write a function to sum the leaves of a binary tree, which has been defined using the free-type notation of Z as follows:

$$\begin{aligned} \mathit{tree} ::= &\ \mathit{Leaf} \\ \mid &\ \mathit{Branch}\langle\!\langle \mathit{tree} \times \mathbb{N} \times \mathit{tree} \rangle\!\rangle \end{aligned}$$

The seven steps are conveniently expressed using an axiom schema:

1. Name the function

   sum

2. Write down its type

   $\mathit{sum} : \mathit{tree} \to \mathbb{N}$

3. Enumerate all cases

   $\mathit{sum} : \mathit{tree} \to \mathbb{N}$

   $\mathit{sum}\ (\mathit{Leaf}) =$ refine
   $\mathit{sum}\ (\mathit{Branch}(\mathit{left}, \mathit{data}, \mathit{right})) =$ refine

4. Deal with any simple case(s)

$$\begin{array}{|l} \textit{sum} : \textit{tree} \to \mathbb{N} \\ \hline \textit{sum}\ (\textit{Leaf}) = 0\ \wedge \\ \textit{sum}\ (\textit{Branch}(\textit{left}, \textit{data}, \textit{right})) = \boxed{\text{refine}} \end{array}$$

5. List the ingredients
   sum, $Branch$, $left$, $data$, and $right$
6. Deal with the complex case(s)

$$\begin{array}{|l} \textit{sum} : \textit{tree} \to \mathbb{N} \\ \hline \textit{sum}\ (\textit{Leaf}) = 0\ \wedge \\ \textit{sum}\ (\textit{Branch}(\textit{left}, \textit{data}, \textit{right})) = \\ \qquad (\textit{sum left}) + \textit{data} + (\textit{sum right}) \end{array}$$

7. Think about the result

To form a proper axiom schema, universal quantifiers would have to be used to introduce the variables. We have chosen to omit these and to rely on appropriate tools [20] to insert the quantifiers and infer the types of the variables.

B.2. Prolog

It is becoming more common for Prolog programmers to explicitly write the types of their predicates. For example Sterling and Shapiro [21, p. 242] write that 'Types are emerging as important in Prolog programs'. Modern commercial implementations, e.g. Visual Prolog [22] and research languages, such as Mercury [23], are also typed. We borrow the notation for types of Mercury to support Programming by Numbers with Prolog.

Problem: Write a predicate to sum the leaves of a binary tree, which has been defined as follows.

0. Lecturer-defined data type
```
:- type tree ---> leaf
              ;    node(tree,int,tree) .
```

The mode declaration below is also written using the Mercury notation but the predicate `sum` is written in Prolog.

1. Name the function(predicate)
   `sum`
2. Write down its type(mode)
   `:- pred sum(tree::in,int::out).`
3. Enumerate all cases
```
sum(leaf, refine ) :-
    refine .
sum(node(Left,Data,Right), refine ) :-
    refine .
```
4. Deal with any simple case(s)
```
sum(leaf,0) .
sum(node(Left,Data,Right), refine ) :-
    refine .
```
5. List the ingredients
   `sum`, `+`, `Left`, `Data`, and `Right`
6. Deal with the complex case(s)
```
sum(leaf,0) .
sum(node(Left,Data,Right),Total) :-
    sum(Left,SumLeft),
    sum(Right,SumRight),
    Total is SumLeft+Data,SumRight.
```
7. Think about the result

The predicate `sum` has two arguments. The first unifies with a `tree` and the second with a number. The mode declaration asserts that the tree must be input, and that a number will be delivered as output. Because of the non-reversibility of the `is` operator, this is the only acceptable moding.

B.3. Pizza

Pizza is essentially Java extended with pattern matching, parametric polymorphism and higher-order methods. Pizza is a proper superset of Java, implemented by a pre-processor. Pizza classes interwork smoothly with Java classes, and thus give all the benefits of working with Java.

Problem: Write a method to sum the leaves of a binary tree, where the tree has been defined as a class `Tree` with two constructors `Leaf` and `Branch`:

```
public class Tree {
    case Leaf;
    case Branch(Tree left, int data, Tree right);
     refine methods
}
```

The method `sum` is able to perform case analysis on the tree, using Pizza's extended notion of switch and case statements:

1. Name the function(method)
   `sum`
2. Write down its type
```
int sum() {
     refine body
}
```
3. Enumerate all cases
```
 switch (this) {
 case Leaf :
      refine
 case Branch(Tree left, int data, Tree right) :
      refine
 }
```
4. Deal with any simple case(s)
```
   case Leaf :
      return 0 ;
```
5. List the ingredients
   `sum`, `+`, `left`, `data`, and `right`
6. Deal with the complex case(s)
```
  case Branch(Tree left, int data, Tree right):
     return left.sum() + data + right.sum() ;
```
7. Think about the result
```
int sum() {
   switch (this) {
   case Leaf :
      return 0 ;
   case Branch(Tree left, int data, Tree right):
      return left.sum() + data + right.sum() ;
   }
 }
```

As with the Java example, we varied the method slightly by using step 7 to assemble the elements created in steps 4 and 6.