Recursion in Gradual Steps (Is Recursion Really that Difficult?)

J. Ángel Velázquez-Iturbide
Escuela Superior de CC. Experimentales y Tecnología, Universidad Rey Juan Carlos
C/ Tulipán s/n, 28933 Móstoles, Madrid, Spain
a.velazquez@escet.urjc.es

Abstract

We propose a gradual approach to teach recursion. Our main assumption is that the difficulty in learning recursion does not come from the recursion concept itself, but from its interaction with other mechanisms of imperative programming. We use this basic idea to propose a new pedagogical approach. On the one hand, recursion is introduced in a gradual way by means of three fields (grammars, functional programming and imperative programming). On the other hand, each instance of recursion is explained so that all of its accompanying mechanisms are clearly identified.

The approach has three main advantages. First, the teaching of recursion is simplified because it is taught in a gradual way. Second, the concept of recursion is isolated and differentiated from other concepts or mechanisms associated to particular instances of recursion. Last, the student perceives recursion as a recurrent concept in the discipline of computer science.

1 Introduction

Recursion is well-known as one of the most difficult topics in computer science education. Being more precise, the difficulty is in the teaching of recursion *in imperative languages*. The main problem is the great complexity caused by its accompanying programming constructs and mechanisms. Sometimes the consequence is a misunderstanding of imperative recursion itself, and others, a misunderstanding of recursion usage.

Many approaches to teaching recursion in imperative languages have been adopted, but none seems to be satisfactory because they do not separate proper recursion from such accompanying features. Some approaches introduce models of recursion for imperative languages while others introduce general models of recursion [5]. Some researchers think that recursion must be explained using abstract models [4], whereas others argue for concrete models [20]. In any case, there is a general consensus in considering recursion a fundamental concept in computer science [15]. Although (paradoxically!) not cataloged in the Computing Curricula 1991 as a recurrent concept, we think that it is. As such, many computer scientists think that it should not be taught as an advanced topic, but it should be taught early [1,7].

We do not ignore the inherent difficulty of the topic, and therefore the difficulty of learning it. However, such a problem has not been identified in other computer science fields where recursion also appears. For instance, non-trivial grammars are recursive, and it does not seem to be a problem for students. Analogously, declarative programming paradigms, such as functional or logic programming, rely heavily on recursion and again it is not perceived as a problem [12,14].

We think that this disparity of difficulties in learning recursion often comes from a confusion of many educators between recursion and imperative recursion. It is not clearly explained that recursion in imperative languages is recursion *plus* several mechanisms, such as parameter passing, control stack, etc. We argue for an approach to the teaching of recursion where all of the mechanisms and concepts involved are clearly identified and isolated from the concept of recursion.

Another source of difficulty for teaching recursion is the complexity of mechanisms that appear in imperative recursion. We argue that the learning curve is smoother if recursion is not explained directly in imperative languages, but in other fields. As a consequence, we adopt a gradual approach, where several instances of recursion are introduced in increasing level of difficulty by studying it in three fields: grammars, functional programming and imperative programming.

In summary, our proposal is two-fold. On the one hand, recursion is introduced in a gradual way by means of its instances in three fields. On the other hand, each instance of recursion is explained so that all of its accompanying mechanisms are clearly identified.

SIGCSE 2000 3/00 Austin, TX, USA

The paper has the following structure. The second section gives a global view to what we call a "gradual approach," by showing examples of its successful application in other computer science fields. The third section describes our gradual approach to teaching recursion, with three subsections corresponding to the three fields we propose to study recursion. Section 4 contains a discussion on our proposal, and in section 5 we summarize our conclusions.

2 Gradual Approaches in Computer Science

There are many topics in computer science which are better addressed with a gradual approach. When we want to design (or analyze) a product, we perform several relatively simple steps. If the problem is difficult and the steps are well chosen, the effort required to make sequentially these steps is qualitatively smaller than solving it in one large step.

The best known representative of this idea is stepwise refinement [19]. This methodology advocates making a program by following a sequence of steps. Initially, a vague program is designed. At each of the following steps, the programmer faces one design decision about program construction. Finally, an executable program is obtained. Notice that not only stepwise refinement has been used for sequential programs, but for concurrent programs as well [17].

A similar idea underlies the methodology of program transformation [9]. The programmer designs a first but usually inefficient version, which after a sequence of transformations is converted into more efficient versions. Each step allows improving the program in a particular concern. Program transformation, together with a collection of "tactics," allows producing non-trivial algorithms in a systematic way, e.g. dynamic programming [3]. The main difference between transformation and stepwise refinement of programs is the number of executable programs produced: with program transformation many equivalent (but not equally efficient) programs are produced, whereas with stepwise refinement only one program is produced (the final one).

In all of these methodologies a central idea lies: complex products are developed more easily in a sequence of small steps than in a single large step. Gradual methods are also being used in computer science education. For instance, there are experiences (e.g. [6]) in programming projects where students are urged to develop programs in gradual steps.

In the paper, we advocate this approach to explain a difficult concept: recursion. We think that recursion is not too difficult by itself. The problem comes from trying to introduce recursion as it is instantiated *in imperative programming languages*. This particular instance of recursion *is* difficult because of its interaction with other mechanisms specific to imperative languages.

A gradual approach can aid in teaching recursion by following several steps. We start with the concept of recursion and later show its occurrence in three fields of computer science. They are introduced in increasing level of difficulty, making explicit the mechanisms which make each instance of recursion more difficult than the previous one. When recursion in imperative languages is explained, the student already has experience with recursion and is prepared to understand it properly.

3 A Gradual Approach to Recursion

We include in this section a gradual approach to learning recursion by means of its instances in three different fields. For each field, we briefly describe the representation of information which is handled recursively, the particular recursive definition and an associated operational model, which explains how recursion is used to perform certain repetitive process. We also make explicit the mechanisms associated to that recursive definition, which are different from recursion itself.

Of course, a previous and simple definition of recursion is in place: a recursive definition defines something in terms of itself.

3.1 Recursion in Grammars

We can affirm that "pure" recursion arises in formal grammars. Well-known examples of recursive grammars describe arithmetic expressions and programming language constructs. Probably the simplest recursive grammar is:

$$S ::= a \mid a\,S$$

that defines the language $\{a^n$, for $n \geq 1\}$.

This simple case of recursion allows illustrating most of the components of well-designed recursive definitions (base case, recursive case), and their problems (non-termination). Let us examine the main concepts involved.

Representation of information. Sentences are strings of terminal symbols. More generally, sentential forms are strings of terminal and non-terminal symbols.

Definition. A grammar rule is recursive if the non-terminal symbol at the left-hand side also appears at the right-hand side.

Operational model. It consists in a form of rewriting, called derivation, that allows proving that a sentence belongs to the language defined by a grammar. We start from the initial symbol and successively apply grammar rules until the target sentence is obtained. For each rewriting, we select a grammar rule and substitute in the last sentential form an occurrence of the rule left-hand side (a non-terminal symbol) by its right-hand side (a sequence of arbitrary symbols).

For instance, the sentence aaa is derived from the previous grammar as follows:

S
→ aS
→ aaS
→ aaa

Notice that there are very few mechanisms associated to rewriting in sentence derivations:

- *Selection.* For each rewriting step, a nonterminal symbol α in the sentential form and a rule with α in its left hand side are selected. In the previous derivation, we have underlined the nonterminal selected in every sentential form (notice that in this example only one nonterminal can be selected at each step!).
- *Substitution.* The sentential form is rewritten by simply replacing an occurrence of the nonterminal symbol by the right hand side of the selected rule.

3.2 Recursion in Functional Programming

We also find recursion in functional programming, where it is usually the only mechanism to perform repetitive computations. The factorial function is the typical example with (again) a base and a recursive case. In ML:

```
- fun fact n =
      if n=0 then 1 else n*fact(n-1);
val fact = fn : int -> int
```

Representation of information. Expressions are syntactic structures of different classes: values, function applications, conditional expressions, etc.

Definition. In a recursive function, the application at the left-hand side of its equation occurs again at the right-hand side, but applied to different arguments.

Some elements of this recursive definition require some attention:

- *Parameters.* The definition of recursive functions involves the use of formal parameters, which are abstract names that represent future actual values. The example contains the formal parameter *n* representing an arbitrary natural number.
- *Well-founded ordering.* A recursive computation must necessarily terminate, imposing restrictions on the form of recursion. Recursive function applications at the right-hand side of the equation must operate on parameters "smaller" than formal parameters at the left-hand side; technically, termination depends on the existence of a well-founded ordering defined over the parameters. In the factorial example, the only recursive application contains the natural number *n* decremented by one.

Operational model. Expression rewriting is the operational model of computation for evaluating functional expressions. An expression is evaluated by successively rewriting it into other expressions until an expression that cannot be further simplified is obtained. The final expression is called the value of the original one. Rules are defined to evaluate each class of expression allowed in the language (function application, conditional expression, etc.)

For instance, the step-by-step evaluation of the factorial of 4 is given by the following sequence of expressions:

```
fact 4
↓
if 4=0 then 1 else 4*fact(4-1)
↓
if false then 1 else 4*fact(4-1)
↓
4*fact(4-1)
↓
4*fact(3)
↓
...
↓
4*(3*(2*(1*fact 0)))
↓
...
↓
24
```

Rewriting in expression evaluation has associated a few mechanisms, including selection and substitution again:

- *Selection.* Given an expression to evaluate, the next subexpression to rewrite (the "redex") is selected, and its associated evaluation rule is applied. In the case of function application, it must be taken into account that functions can be defined by several equations, so the equation to be applied must also be selected.

  In the previous evaluation, we have underlined every redex. In the factorial function there is only one rule, but in functions defined with several equations, a more elaborate selection mechanism, called *pattern matching*, is provided. For brevity, we do not explain it here.
- *Parameter binding.* Before performing substitution, binding is established between formal parameters and actual parameters. In the example, *n* is bound to 4. In more complex functions, binding is a side effect of selecting the adequate equation by pattern matching.
- *Substitution.* The redex is replaced by a new expression. In the case of function application, it is replaced by the right-hand side of the selected equation, where formal parameters have been previously substituted by actual parameters. Thus, in the previous example,

  `fact 4`

  was replaced by

  `if 4=0 then 1 else 4*fact(4-1)`

3.3 Recursion in Imperative Programming

Finally, recursion in imperative programming is taught as a mechanism complementary to loops in order to perform repetitive computations. We show its use by means of a procedure that computes the factorial function (we use a procedure rather than a function for generality). In Pascal:

```
procedure fact (n: integer;
                var f: integer);
  begin
    if n=0 then
      f := 1
    else begin
      fact (n-1, f);
      f := n*f
    end
  end;
```

Representation of information. Information is stored in abstractions of memory cells called variables. Pointer variables are especially remarkable because they do not contain values, but memory addresses (of other variables). Most memory cells are declared explicitly by the programmer (variables), but others exist in run time without declaring them (the control stack).

Definition. A recursive procedure contains some call of itself in its body, but applied to different arguments.

We find again parameters and well-founded orderings in recursive procedures. In addition, parameters have the following feature:

- *Parameter modes.* Each formal parameter has associated a passing mode, typically either by value or by reference. Each of these modes determines the kind of actual parameter that can be used. Thus, the actual parameter for a parameter by value can be any imperative expression, but for a parameter by reference, it must be a variable.

Operational model. The operational model of procedure call is associated to the operational model of imperative programming defined as state transformation. The current state of execution is given by the contents of variables plus information about the next statement to execute (stored in the program counter).

A procedure call is a control break with return. In addition, the set of accessible variables changes. The main concepts necessary to understand all of these changes are:

- *Parameter binding.* Depending on the mode of each parameter, a different binding is established between formal and actual parameters. For instance, a parameter passed by value is an expression that is first evaluated, then a new local variable is created and finally it is assigned that value. However, for a parameter passed by reference, a pointer is allocated to point to the actual parameter variable.
- *Activation records and the control stack.* An activation record is a structure invisible to the programmer but necessary to start and finish procedure calls. It keeps information necessary to execute a procedure and later restore execution in the statement following the procedure call. This information is composed at least by actual parameters, local variables and the return address.

  The control stack is the structure that contains activation records during program execution. It is a complex structure designed to store all of the information necessary to make procedure calls, with some intricacies to handle names collisions.

4 Discussion and Related Work

In the academic year 1997/98 we began to teach a course on principles of programming languages for freshmen. The course is atypical because similar courses are offered in other universities to junior or, at least, sophomore students. This timing forced us to design a course with innovative approaches to many topics, including the gradual approach to recursion we have described in the paper.

The previous introduction to recursion was mostly driven by computations. A different line of exposition can be followed by focusing on data types. The similarity of recursion in grammars, functional data types and imperative data types is even greater than in recursive control, so the exposition is easier. However, its main drawback is the requirement to know pointers.

We do not know of people who use our method to teach recursion, although there are similar, less ambitious proposals. Richard Pattis [10] proposes to teach EBNF in the first lecture of CS1. EBNF is simple but rich enough to illustrate a set of computer science concepts, including recursion. Mary Shaw et al. [13, p.65] also had mentioned this possibility. Both proposals introduce recursion using EBNF, but they do not mention how to make use (not even if they do) of this knowledge in order to teach recursion in imperative languages.

Nowadays, there are many experiences in the use of functional programming as a first language [12,14]. However, the transition from functional programming to imperative programming usually does not address explicitly the similarities and differences of recursion in both paradigms. Besides, they do not include the prologue about recursion in grammars, which I find very convenient.

We are aware that our proposal is unrealistic for most CS1/CS2 programs, but it can be adopted, limited to grammars and imperative programming. From a pedagogical point of view, even this restricted approach participates and benefits from the sequencing, spacing and spiraling methods [11]. In effect, we propose a particular sequencing of several instances of

recursion. Consequently, recursion is spaced out over several months and it is revisited several times.

Notice that our approach only provides a framework to teach recursion in a gradual way. However, the approach is orthogonal to debates about how to teach certain instances of recursion, most notably imperative recursion. We simply require to identify key concepts and mechanisms, but they can be taught in many different ways. However, we want to make two recommendations from our experience:

- A problem with explaining recursion is that there are many other concepts which can arise, although they are not exclusive of recursion. Some examples are: scope of identifiers, subprogram pre- and postconditions, etc. We recommend the teacher to teach them first, so that the subsequent explanation of recursion will not be darkened by these details.
- The explanation of every instance of recursion can benefit from visualization facilities, such as derivation trees for grammars. Elsewhere can be found visualizations of sentence derivations [2], evaluation of functional expressions [18] and control stack issues [8,16].

5 Conclusions

We have proposed a gradual approach to simplify the teaching of recursion. It is introduced in increasing level of difficulty by means of three occurrences of recursion in different computer science fields, namely formal grammars, functional programming and imperative programming. The approach has three main advantages. First, the student perceives recursion as a recurrent concept in the discipline of computer science, not only existing in imperative programming. Second, the concept of recursion is isolated and differentiated from any other concept or mechanism associated to a particular instance of recursion. Finally, the teaching of recursion is simplified because it is taught in a gradual way.

6 Acknowledgments

I want to thank the valuable comments of the anonymous referees. This work was supported by the Comunidad Autónoma de Madrid under project no. 07T/0036/98.

References

[1] Astrachan, O., "Self-reference is an illustrative essential," *25th SIGCSE Technical Symposium on Computer Science Education*, 1994, pp. 238-242

[2] Bilska, A. O., et al., "A collection of tools for making automata theory and formal languages come alive," *28th SIGCSE Technical Symposium on Computer Science Education*, 1997, pp. 15-19

[3] Bird, R., "Tabulation techniques for recursive programs," *ACM Computing Surveys*, Vol. 12, No. 4, December 1980, pp. 403-417

[4] Ginat, D., Shifroni, E., "Teaching recursion in a procedural environment– How much should we emphasize the computing model?," *30th SIGCSE Technical Symposium on Computer Science Education*, 1999, pp. 127-131

[5] Haynes, S.M., "Explaining recursion to the unsophisticated," *SIGCSE Bulletin*, Vol. 27, No. 3, Sept. 1995, pp. 3-6 and 14

[6] Leeper, R., "Gradual project assignments in computer courses", *20th SIGCSE Technical Symposium on Computer Science Education*, 1989, pp. 88-92

[7] McCracken, D.D., "Ruminations on computer science curricula," *Communications of the ACM*, Vol. 30, No. 1, January 1987, pp. 3-5

[8] Naps., T.L., Stenglein, J., "Tools for visual exploration of scope and parameter passing in a programming languages course," *27th SIGCSE Technical Symposium on Computer Science Education*, 1996, pp. 305-309

[9] Partsch, H.A., *Specification and Transformation of Programs*, Springer-Verlag, 1990

[10] Pattis, R.E., "Teaching EBNF first in CS 1," *25th SIGCSE Technical Symposium on Computer Science Education*, 1994, pp. 300-303

[11] Powers, K.D., D.T. Powers, "Making sense of teaching methods in computer science", *1999 Frontiers in Education Conference*, session 11b3

[12] Reinfelds, J., "A three paradigm first course for CS majors," *26th SIGCSE Technical Symposium on Computer Science Education*, 1995, pp. 223-227

[13] Shaw, M., et al. (eds.), *The Carnegie-Mellon Curriculum for Undergraduate Computer Science*, Springer-Verlag, 1985

[14] Thomson, S., Wadler, P. (eds.), monographic issue on "Functional programming in education," *Journal of Functional Programming*, vol. 3, n°. 1, January 1993

[15] Tucker, A., et al, *Computing Curricula 1991*, ACM Press and IEEE Computer Society Press, 1991

[16] Velázquez-Iturbide, J.A., "Formalization of the control stack," *SIGPLAN Notices*, Vol. 24, No. 3, March 1989, pp. 46-54

[17] Velázquez-Iturbide, J.A., "A methodology for monitor development in concurrent programs," *SIGCSE Bulletin*, Vol. 26, No. 3, September 1994, pp. 22-28

[18] Velázquez-Iturbide, J.A., A. Presa-Vázquez, "Customization of visualizations in a functional programming environment," *1999 Frontiers in Education Conference*, session 12b3

[19] Wirth, N., "Program development by stepwise refinement," *Communications of the ACM*, Vol. 14, No. 4, April 1971, pp. 221-227

[20] Wu, C.-C., Dale, N.B., Bethel, L.J., "Conceptual models and cognitive learning styles in teaching recursion," *29th SIGCSE Technical Symposium on Computer Science Education*, 1998, pp. 292-296