The Structuring Power of Mathematica in Mathematics and Mathematical Education

R. Barrere, University of Franche-Comte, ENSMM
26 chemin de l'Epitaphe, F-25000 Besancon, France
E-mail: rbarrere@ens2m.fr, Fax: 33(0)381 809 870

■ Abstract

So far, Mathematica has been used mainly either in teaching as a support for classical courses, or in research as a computing environment. However, it could be assigned a wider and more profound role in mathematics and mathematical education. In particular, it could support, facilitate and accelerate the integration of algorithmic aspects into the corpus of mathematics.

■ 1 Introduction

The necessity of integrating computing aspects in scientific education has already been expressed ([6] for instance) and some authors describe innovative experiments based on the use of Mathematica ([7] for instance). However, other authors stress the necessary caution in the renovation of courses or curriculums [5]. The following discussion was stimulated by the observation of a general trend "to bend Mathematica to old-fashioned pedagogic strategies that leave much of its potential unexplored" [17]. To a wide extent, this remark also applies to research where the high-level symbolic potential of Mathematica is often underused. One of the reasons might be a common belief that mathematical knowledge is unchanging, with unending concepts. Actually, concepts evolve, even the most fundamental ones.

Nevertheless, as a result of the availability of processing power, a renewed way of doing and teaching mathematics is emerging, which stresses its algorithmic aspects besides logic. Mathematica could play a central role in this respect. At an elementary level, the package has already been used as a platform to initiate students into computer algebra, numerical methods and elements of programming. At a more advanced level and in research, it could constitute a common language to tackle such intermingled questions as algorithmic mathematics, theories of computation, program description and design, modelling and simulation methods.

The paper is partly based on an experiment with students in engineering. It began with the use of Mathematica as an integrated tool for training them in computer algebra, numerical methods and symbolic programming plus a little mathematical modelling; it was based on computer labs with personal reports. The success of this first experiment incited the training personnel to bet on the federative power of Mathematica and to extend this approach to lectures and tutorial workshops. Now, a presentation of language theory, a discussion about programming paradigms, an overview of the main programming constructs and an initiation into program proving are part of a few introductory lectures beside logic. This novel experiment is in progress and plans already exist for broadening it to an introduction to complexity analysis, logical programming and mathematics for computer algebra.

Some of these topics are already taught in computer science courses ([4, 10] for instance); others are still scattered about the literature on theoretical computer science. However, the computer has become such a universal tool that elements of computer science must now be part of every scientific education. One of the purposes of the following informal presentation is to convince the reader that these topics deserve to be treated from a mathematical point of view in the same way as set theory or logic.

■ 2 Computing vs reasoning

The sequence defined by v[0]=1; v[n_]:=n-v[v[n-1]] gives a simple illustration of the difference between the classical mathematical reasoning, which yields v[1]=1/2 by solving the equation v[1]==1-v[1], and the computational approach which leads to an infinite recursion. Quite surprisingly, colleagues like students, whom the answer "infinite recursion" leaves skeptic, answer v[1]=1/2. This common reaction shows that, as a result of their classical education, many scientists are still unaware of algorithmic questions.

Despite this surface discrepancy, further thought shows that the two approaches are not incompatible for the question can be restated from the enlarged point of view: which rule must be applied and when (i.e., in which order rules must be applied)? Several examples can be given about the role of rule precedence during evaluation, both in Mathematica and in classical mathematics. For instance, with Mathematica, upvalues are tried before downvalues, which allows overriding other rules; similarly, the attributes of the family "hold" enable non-standard evaluation, in particular the localization of variables; in mathematics, a variational formulation consists of an integration by parts before any other evaluation.

The example also brings out the difference between the transformational interpretation of the sign = (= or := in Mathematica) and the equational one which is relational. The transformational formulation is single-valued ("deterministic"), whereas the relational one is multivalued; it is inopportunely said "non-deterministic" although it doesn't violate classical determinism. Basically, a computation is a transformational process while a reasoning uses relational formulations.

The example further stresses the distinction between rules that are applied automatically (the recursive definition) and those that the user must explicitly apply (Solve[v[1]==1-v[1]]). Finally, it emphasizes the importance of program proving, in particular termination proofs and the role of structural induction.

So this discussion points out the rule-based paradigm as a basic tool to tackle mathematical questions from the computational point of view and symmetrically computational questions from the mathematical point of view. In order to go deeper into the matter, two entangled questions must be answered first: how mathematical expressions (and computer programs) are written, and how they are processed. As a matter of fact, expressions are generated by context-free grammars and computations are implemented by (or amount to) a rewrite (context-sensitive) mechanism.

■ 3 Language theory and syntactic questions

3.1 Elements of language theory

It might be a truism that mathematical statements must be written so that mathematics relies upon language theory, even prior to logic. However, this seems not to have been recognized until recently (actually, linguistic questions have often been improperly identified with logical ones), so that investigations about languages began under the patronage of linguistics and computer science rather than mathematics. Moreover, many authors now stress the algebraic structure (semi-group or monoid) of languages although their primary interest is to understand the underlying branching structure of mathematical formulas and computer programs.

Although they initially emerged on the occasion of Chomsky's investigations [2] about natural languages, mathematical ideas about languages were mainly developed in the context of programming. The central question is to characterize (generate or recognize) well-formed expressions among all possible sequences of symbols. This role is devoted to grammars (i.e., sets of production rules) and recognizers or parsers. In the Chomsky hierarchy, two major classes are of interest for language design: regular languages describe the sequential structure of codes, words, numbers or strings (atomic expressions in Mathematica); context-free languages describe the branching structure of mathematical expressions or programs (non-atomic expressions in Mathematica). Basically, mathematics as well as computer science rely upon context-free languages.

In a programming context, it would be necessary to introduce automata theory in view of the design of recognizers or parsers. In a more general scientific context, in order to avoid the cumbersome manipulation of automata, one can analyse strings by "exploding" them into lists and by using recursive operators [12]. Here is for instance, a predicate that tests whether a string over the alphabet ={(,)} is a well-formed bracketing (Dick's language).

```
explode[s_String] := Map[FromCharacterCode, ToCharacterCode[s]]

DickQ[s_String] := DickQ[explode[s]]

DickQ[{}] := True (* empty word *)
DickQ[{"(", w___ /; DickQ[{w}], ")"}] := True
DickQ[{u__ /; DickQ[{u}], v__ /; DickQ[{v}]}] := True
DickQ[___] := False

Map[DickQ, {"(()())()", "(()", "", "[]"}]

{True, False, True, False}
```

The recursive predicate is a direct implementation of the set of production rules {S ε, S (S), S SS} (where S is the start symbol and ε the empty word) or of the equivalent logical description: ε L, w L (w) L, u L∧v L uv L (where L denotes Dick's language).

3.2 Functional syntax and abstract syntax

Behind these notions, there is a central question: languages are a means of representing intrinsically branching structures by sequential ones. A small number of fundamental methods to represent trees by sequences have been put forward: the prefix, infix and suffix notations plus occasionally the matchfix one. Since these notations are different ways to represent a tree, which is the underlying significant structure, they are equivalent and it can be more attractive to reason directly with trees and their associated terminology; this is the role of abstract syntax.

So far, these different writings (concrete syntaxes) have mainly be used for convenience. In fact, for better efficiency in communication, it would be better to use a single notation. The designers of Mathematica used the ergonomic solution of a functional syntax, which is a combination of prefix and matchfix notations. This internal form (FullForm) of expressions has been supplemented with a variety of input and output formats to ensure compatibility with classical mathematical notations. Since version 3, these formats include the mathematical two-dimentional graphical notations. Below, a few functional translations from logical, algebraic, Lisp, HTML or PostScript statements bring out the unifying and often clarifying power of the functional syntax.

Domain specific	Functional
(P ∨ Q) ∧ R	And[Or[P, Q], R]
x + 3 y + sin π	Plus[x, Times[3, y], Sin[Pi]]
(+ (* x (log x)) x)	Plus[Times[x, Log[x]], x]
< \head > The title < \Head >	Head["The title"] or Cell["The title", "Head"]
0 0 moveto 1 1 lineto	Line[{{0, 0}, {1, 1}}]

Thus, the functional syntax can be viewed as a conventional abstract syntax. Indeed, there is a close relation (a morphism) between trees and functional constructs. This was a great idea to choose a functional notation at the heart of Mathematica, for it is usual and has a strong mathematical flavor. The figure below shows the functional and "applicative" representations of the expression $e_0[e_1,e_2\ldots e_n]$.

If need be, the formatting of expressions can constitute application exercices. Here is an example with the format prefixListForm that gives the prefix Lisp-like form of an expression.

```
prefixListForm[e_?AtomQ]:= e
prefixListForm[h_[p___]]:= Map[prefixListForm, {h, p}]
```

```
prefixListForm[f[a,g[1,1.5],h[]]]
```

```
{f, a, {g, 1, 1.5}, {h}}
```

3.3 Functional syntax and structural induction

The algorithmic approach involves sets defined by structural induction. Structural induction is an extension of classical induction to partial orderings. In practice, it allows generalizing proofs from sequential structures to branching ones. Indeed, in the context of a functional syntax, objects (expressions in Mathematica) are viewed as the result of applying functions to other objects so the resulting functional expressions are trees. As a consequence, program proving involves structural induction. Here is an example with a recursive definition for the depth of an expression.

```
depth[e_?AtomQ] := 1
depth[_[]] := 2
depth[e_] := 1 + Apply[Max, Map[depth, e]]
```

One can verify that the relation "is a subexpression of" determines a well-founded partial ordering. Then, the property that the function terminates and yields the correct result is proved by structural induction in two steps. Firstly, the property is clearly true for atoms, which are the minimal elements for the well-founded ordering and the base cases for the recursion. Then, assuming the property is true for the expressions of depth (strictly) lower than n, one concludes it is also true for expressions of depth n since Map[depth, e] only brings into play expressions of depth lower than n. In more abstract terms this amounts to proving that the computation generates a decreasing sequence in a well-founded set.

Behind the principle of structural induction, there is the idea that membership of a set can be described in term of the form of the object. In Mathematica, theses forms are characterized by patterns, that can be viewed as expression templates or classes, including logical tests (see section 5 below). Classical mathematics lays stress upon the structure of sets, i.e. the collection of relations between unstructured objects; on the contrary, the algorithmic point of view stresses the internal structure of objects.

■ 4 Programming paradigms and theories of computation

An algorithm describes the decomposition of a computation into elementary operations. There are several ways of expressing algorithms according to these elementary operations, i.e., the primitives of the language. According to whether the language is machine oriented or problem oriented, this leads to several sorts of languages supporting different programming paradigms. [16, 18]

Three major programming paradigms turn out to have three main theories of computation as mathematical counterparts: Post-Turing machines for procedural programming, Kleene's recursive functions and the Church -calculus for the functional paradigm and the Post-Markov theory of algorithms for the transformational paradigm. So we have the equation: programming paradigm model of computation. Let's examine them more precisely.

So far, the procedural paradigm has been the most widespread one. Strongly influenced by the most usual computer architecture, it identifies the processing device with an automaton evolving through successive states. Post-Turing machines constitute a low-level model of the procedural paradigm and Fortran (Backus), Pascal (Wirth) or C (Kernighan) are well-known implementations of it.

```
proceduralMagnitude[v_] :=
    Module[{x, y},
            x = v[[1]]; y = v[[2]];
            z = Sqrt[x^2 + y^2];
            Print[z] ]
```

However, other programming models were imagined, such as the functional paradigm based on the idea of mathematical evaluation: since a function gives a result (output) for any admissible value of the variable (input), it constitutes a model of computation. Lisp (Mc Carthy) was the first example of such a language, which is said functional (or applicative). It consists of mechanisms to define and compose functions and apply them to values. The theory of recursive functions and the -calculus constitute mathematical counterparts of this programming paradigm.

```
functionalMagnitude[v_] :=
        Sqrt[First[v]^2 + Last[v]^2]
```

Another non-conformist approach is based on the fact that data are written, so an algorithm consists in analysing and transforming sequences of symbols according to specified rules. Such a syntax-oriented language is called transformational (or rule-based) and Snobol (Griswold) is an actual implementation. It consists of a comparison process called pattern-matching associated with a rewrite mechanism. Post's formal systems or Markov's theory of algorithms constitute mathematical counterparts of this programming paradigm.

```
transformationalMagnitude[{x_, y_}] :=
        Sqrt[x^2 + y^2]
```

These questions about programming styles are far from useless. The issue was clearly stated by Budd [1]: "the language in which a programmer thinks a problem will be solved will color and alter, in a basic fundamental way, the fashion in which that programmer will develop an algorithm." Hence the interest in a rich enough linguistic frame so that different programming styles (or models of computation) may coexist or even cooperate: "I knew at that time that the real revolution would arrive when we finally understood how to bring these diverse points of view together in one framework" [1]. The idea of multiparadigm programming emerged in the early eighties; Leda was Budd's first attempt to implement a multiparadigm language, after which Mathematica was the first widespread multiparadigm language, built on top of a rule-based core with a functional syntax. Then, computing consists in rewriting functional expressions, i.e., rewriting trees.

Plenty of exercises can be made to illustrate paradigm conversions. One generally admits that the functional and transformational styles are the most elegant ones, except in the case of compilation for numerical algorithms. So these exercises will generally read: "transform this procedural program into a functional one", or: "convert this functional definition into a transformational one". For instance, the procedural program:

```
aSum[myList_] := Module[
        {theSum = 0},
        Do[theSum = theSum + myList[[i]]^2,
            {i, 1, Length[myList]}];
        theSum
        ]
```

which squares the elements of a list and adds them would give, thanks to the listability of Plus:

```
aNicerSum[myList_] := Apply[Plus, myList^2]
```

Similarly, the functional definition:

```
thisToCartesian[v_] := First[v] { Cos[Last[v]], Sin[Last[v]]}
```

which converts a polar representation to a cartesian one, would give:

```
thatToCartesian[{r_, θ_}] := r { Cos[θ], Sin[θ]}
```

Finally, the relational (e.g., Prolog) and object-oriented (e.g. SmallTalk) paradigms could be mentioned. They are not directly implemented in Mathematica, but Meader [14] described a simplified implementation of the relational paradigm in Mathematica and several object-oriented features of the language deserve a mention: expressions are objects (without internal state), patterns determine classes, upvalues work like methods.

In most scientific applications, computer science has suffered the lack of a common and efficient language to describe algorithms, write programs and reason about them. A great deal of energy has been expended translating problems into programs, programs into mathematical descriptions, defining various pseudo programming languages to describe algorithms, or converting programs from one language to another, while this energy would have been used more efficiently to discuss models, algorithms and data representations. Mathematica clearly avoids these drawbacks by providing with a single consistant framework.

■ 5 Math-oriented constructs vs machine-oriented constructs

5.1 Basic programming constructs

Although the classical mathematical approach rests on set theory and logic, the computational one is rather based on algorithms. Two major programming constructs are iteration and conditionals. Elementary primitives are assumed to be available, such as function definition, composition and application. There are two kinds of iteration: the parametrized iteration corresponds to primitive recursive definitions and according to the programming paradigm can be implemented by means of Do, Fold or a recursive definition that terminates; the conditional iteration corresponds to general recursive definitions and can be implemented by means of While, FixedPoint, an iterative definition or possibly a stream.

In the following example, we seek the first positive integer that equals the sum of its strict divisors; the three programs implement a conditional iteration and yield 6. At present, the implementation formerly said "terminal recursive" is rather said "iterative". Actually, we should differentiate between "self-referent" and "recursive".

```
perfectQ[n_Integer] := TrueQ[n == Apply[Plus, Drop[Divisors[n], -1]]]

(* procedural implementation *)
n = 1; While[Not[perfectQ[n]], n++]; n

(* functional implementation *)
FixedPoint[(#1 + 1) &, 1, SameTest -> (perfectQ[#2] &)]

(* transformational implementation *)
perfect[n_?perfectQ] := n
perfect[n_] := perfect[n + 1]
perfect[1]
```

These constructs apply to lists or more generally sequential data structures, i.e., totally ordered sets from the mathematical point of view. They can be extended to branching data structures, i.e., partially ordered sets. In that case, MapAll gives the iterative counterpart of a recursive definition.

```
recursiveLeafCount[e_?AtomQ] := 1
recursiveLeafCount[e_] := recursiveLeafCount[Head[e]] +
        Apply[Plus, Map[recursiveLeafCount, e]]

visitor[e_?AtomQ] := 1
visitor[e_] := Head[e] + Apply[Plus, e]
iterativeLeafCount[e_] := MapAll[visitor, e, Heads -> True]
```

Programs that implement recursive definitions may not terminate so a termination proof is required. It turns out to be simpler in the case of functional programs, since the proof (by weak, strong or structural induction) is modelled on the recursive definition (see section 3.3). In the near future, program proving, in particular termination proofs will tend to become part of many algorithm-oriented exercises, in the same way as series convergence in calculus exercises.

Finally, conditionals are implemented with If, Which, Switch or by means of conditional definitions. A conditional is equivalent to a search in an ordered set of transformation rules. In another words, the mathematical counterpart of a conditional is a piecewise function.

```
proceduralFactorial[n_] := If[n == 0, 1, n proceduralFactorial[n - 1]]

functionalFactorial[0] := 1
functionalFactorial[n_] := n functionalFactorial[n - 1]
```

In all cases, the comparison of the different styles brings to light equivalences between programming constructs and mathematical formulations. Functional or transformational constructs are naturally math-oriented whereas procedural ones are machine-oriented. We should recommend students to prefer the former, except in particular circumstances such as compilation.

5.2 Advanced programming constructs

The aforementioned constructs are only basic ones, from which any algorithm can be implemented, at least in principle. However, the functional and transformational paradigms introduce specific operators for particular tasks, for instance Nest, Map, Dot … These high level versatile constructs lead to programs that mimic mathematical formulas.

These operators are generic insofar as they apply to categories of problems rather than particular problems. Then, the art of programming consists in choosing, combining or instanciating appropriate operators. This leads to mathematical constructs that facilitate the reasoning about programs. For instance, one can recognize a generalized tensor product behind r-permutations.

```
rPermutations[myList_List, n_Integer] :=
 Flatten[Outer[List, Apply[Sequence, Table[myList, {n}]]], n - 1]

rPermutations[{a, b, c}, 2]

{{a, a}, {a, b}, {a, c}, {b, a}, {b, b}, {b, c}, {c, a}, {c, b}, {c, c}}
```

Once again, it is easy to elaborate lots of little exercises to compute tables, scalar products, differential operators… Classical programming exercises ask for procedural solutions; it would be more enriching to incite students to solve exercises using several programming paradigms. For those who have already learned a procedural language, there are nice exercises about the conversion of procedural programs to functional or transformational ones (see section 3). In the literature on computer science, one also finds exercises about the transformation of recursive difinitions into iterative ones; here is an exemple.

```
recursiveQuicksort[{}] := {}
recursiveQuicksort[{e_}] := {e}
recursiveQuicksort[{p_, r___}] :=
        Flatten[{
                recursiveQuicksort[Select[{r}, OrderedQ[{#, p}] &]],
                {p},
                recursiveQuicksort[Select[{r}, Not[OrderedQ[{#, p}]] &]]
        }, 1]

iterator[{}] := {}
iterator[{e_}] := {e}
iterator[{p_, r___}] :=
        Sequence[Select[{r}, OrderedQ[{#, p}] &],
                {p}, Select[{r}, Not[OrderedQ[{#, p}]] &]     ]
iterativeQuicksort[aList_] :=
        Flatten[FixedPoint[Map[iterator, #] &, {aList}], 1]
```

■ 6 Towards algorithmic mathematics

Computer science brings out the algorithmic approach to function definition, in contrast to the classical set-thoretic approach. Firstly, a function is defined constructively as a combination of primitives or previously defined functions. So a function is viewed as a computing tool rather than a relation between elements of sets. Then, thanks to patterns, function domains (algorithm preconditions) can be incorporated into the definitions.

Moreover, this algorithmic approach brings to light there are generally several ways of doing things, i.e., several possible right-hand members in a definition. This leads to differentiate between the specification of a function and its implementation. The former more or less corresponds to the classical set-theoretic definition; actually, at first stage, it is often given informally in natural language. The latter corresponds to the constructive definition. That's a matter of programming art to find an implementation that satisfies compactness or efficiency constraints; hence the role of complexity analysis.

6.1 Patterns and the modelling of data

A pattern characterizes the class of expressions that match it, so patterns are a means of dividing up expressions into classes; that's the way a pattern denotes an invariant of a class. These classes are de facto organized in subclasses, which establishes a relation with set theory and with object-oriented programming. However, they need not be declared; they are rather used to specify the domain of functions. In practice, patterns constitute a convenient notation for describing subsets of expressions. Using patterns turns out to be convenient to refer to classes, by writing for instance _Real?Positive for the class of positive decimal numbers; then, _ denotes the more general class, i.e., the set of all well-formed expressions.

Actually, one can focuss on the head of an expression, which amounts to a classical type, i.e., a nominal type (e.g., Rational, List, Graphics). Thanks to patterns, one can also consider the structure of an expression, in which case one can think of a structural type. Finally, one can associate a predicate or a test P with a pattern (_/;P or _?P) which then denotes a logical type. In particular, any user-defined predicate can be reused in a pattern, which allows defining recursive classes or enumerated classes.

```
binaryTreeQ[e_?AtomQ] := True
binaryTreeQ[h_?AtomQ[l_?binaryTreeQ, r_?binaryTreeQ]] := True
binaryTreeQ[___] := False

dayQ[x_] := MemberQ[{Sunday, Monday (* … *), Saturday}]
```

Moreover, patterns can be nested so that any combination of these type constructs also charaterizes a class. In particular, one can construct intersections and unions of classes; for instance, _head?test|pat corresponds to the set of expressions (_head ⋂ _?test) ⋃ pat. As a consequence, patterns turn out to be a finer and more versatile way of differentiating data than the classical nominal types.

6.2 Patterns and the modelling of algorithms

One of the main uses of classes is the specification of function domains (algorithm preconditions). Patterns constitute an elegant way to integrate the domain into the function definition. Then, the precondition is automatically tested at the time of function application. Besides, any functional program amounts to function composition, hence a mathematical expression. So proving a program amounts to demonstrating a mathematical property and a correct program is no more no less than a sound formula. In other words, a bug has the same statute as a mathematical mistake.

In practice, proving a function amounts to examine the right hand side of its definition and imagine its evaluation mentally. In particular, $e_0[e_1,e_2\ldots e_n]$ terminates provided that each e_i terminates. Functional or transformational programs lead to direct proofs in general and inductive proofs in the case of recursive definitions. On the contrary, procedural programs with lots of assignments and local variables are unclear and lead to intricate proofs. They require not only to follow the program but also the numerous internal states associated with it.

A peculiarity of algorithms is genericity, which means algorithms apply to classes of data rather than particular data. Patterns are precisely the way to specify classes, consequently to express genericity, hence a mathematical abstraction. In practice, an ordinary definition renders a logical equivalence, whereas a conditional one conveys an implication.

```
evenQ[n_] := evenQ[n - 2] (* evenQ[n-2]  evenQ[n] *)

evenQ[n_] := True /; evenQ[n - 2] (* evenQ[n-2] ⇒ evenQ[n] *)
```

6.3 Operational types

Attaching transformation rules to classes gives abstract data types [13], which more or less correspond to objects in object-oriented programming. Actually, these types were abstract in the procedural context, because they were used to develop mathematical models of data structures and programs. In a functional context, they are rather "operational" and typically implemented by means of upvalues in Mathematica. Then, the old procedural equation "algorithms + data structures = programs" tends to become:

transformation rules + classes $\simeq$ abstract (operational) types $\simeq$ objects.

Finally, one must notice that abstract types are equational, while rules are transformational. So implementing an abstract type amounts to choosing an orientation for equations. The issue was tackled by Maeder in [15]. This remark sets the borderline between mathematics and computer science. Moreover, in mathematics, we are not concerned about the attachment of rules to objects, while this question is central in a programming context. Once again, we can note that computer science enriches mathematics.

■ 7 Discussion

The presentation successively focussed on syntactic questions, programming paradigms, program constructs and program proving methods. It incited to think in terms of trees or functional expressions (abstract syntax) rather than words or formulas (concrete syntaxes), transformation rules and functions rather than programs, evaluation rather than execution, patterns and classes rather than types.

Nevertheless, it could also include more advanced topics such as algorithmic information theory [3], complexity analysis [19] or mathematics for computer algebra [11]. Besides, according to the context (teaching or research), these ideas could be more or less detailed, developed or complemented. For instance, the following topics could be developed: -calculus, confluence properties, formal systems, the relational paradigm, the variety of variables, the problem of assignment and knowledge evolution…

There is basically nothing novel, except slight modifications in the way of presenting notions, a different organization of concepts and hopefully a better integration of mathematical and algorithmic aspects. Set theoretic (relational) and algorithmic (transformational) aspects of mathematics turn out to be complementary, so they should now be examined together.

We could notice the federative power of Mathematica both in computer science and mathematics. It contributes to broadening the zone of influence of mathematics and to bringing novel questions there, such as the dynamical aspects of programming (evaluation process, knowledge growth).

Scientific revolutions seem abrupt at the scale of history, but they are progressive at the scale of human life, because they consist of slight successive changes. The unifying functional syntax and the unitary evaluation mechanism of Mathematica might constitute a revolution at the common borderline of mathematics and computer science; but this is not clear yet because we lack the experience of history.

■ 8 Conclusion

So far, the algorithmic aspects of mathematics have been more or less regarded as marginal in science and scientific education. However, as a result of the common use of computers, they have become central and should be given a better place. Thanks to its functional syntax that mimics mathematical notations, Mathematica turns out to be a good candidate to express and investigate the mathematical aspects of computer science and the algorithmic aspects of mathematics; it brings together the mathematical and computational approaches. With Mathematica, computer science has taken a step towards mathematics; isn't it time mathematics made a move in the direction of computer science?

■ References

[1] T. Budd: Multiparadigm Programming in Leda, Addison-Wesley, 1995

[2] N. Chomsky: On certain formalproperties of grammars, Information and Control, 2(2), 1959, 137-167

[3] G. Chaitin: Algorithmic Information Theory, Cambridge University Press, 1987

[4] J. Gersting: Mathematical Structures for Computer Science, Computer Science Press, 1993

[5] J. Gregor: Computers and active learning of mathematics, in Proceedings of the 9th European Seminar on Mathematics in Engineering Education, Arcada Polytechnic, 1998, 52-57

[6] A. Hayes: Mathematica and people - making the future, in [8], 1-6

[7] D.R. Johnson, J.A. Buege: Rethinking the way we teach undergraduate physics and engineering with Mathematica, in [8], 233-241

[8] V. Keränen, P. Mitic (Ed): Mathematics with Vision, Proceedings of the First International Mathematica Symposium, Computational Mechanics Publications, 1995

[9] V. Keränen, P. Mitic, A. Hietamäki (Ed): Innovation in Mathematics, Proceedings of the Second International Mathematica Symposium, Computational Mechanics Publications, 1997

[10] R. Graham, D. Knuth, O. Patashnik, Concrete Mathematics, Addison-Wesley, 1989

[11] S. Czapor, K. Geddes, G. Labahn: Algorithms for Computer Algebra, Kluwer Academic Publishers, 1992

[12] R. Leermakers: The Functional Treatment of Parsing, Kluwer Academic Publishers, 1993

[13] R. Maeder : The Mathematica Programmer, Academic Press, 1994

[14] R. Maeder : The Mathematica Programmer II, Academic Press, 1996

[15] R. Maeder : Term rewriting and programming paradigms, in [8], 7-19

[16] R. Maeder : The Design of the Mathematica Programming Language, Dr Doob's Journal (April 1992), 86-97

[17] P. Ramsden : Mathematica in Education: Old Wine in New Bottles or a Whole New Vineyard? in [9], 419-426

[18] M. Swaine: Stephen Wolfram, Multiparadigm Man, Dr Doob's Journal, 18 (January 1993 109-112 and February 1993 105-108)

[19] J. Van Leeuven, Ed: Handbook of Theoretical Computer Science, vol A, Algorithms and Complexity, Elsevier, 1992