Metaprogramming and Free Availability of Sources

Two Challenges for Computing Today*

François-René Rideau Đặng-Vũ Bân
francoisrene.rideau@cnet.francetelecom.fr
http://fare.tunes.org/

CNET DTL/ASR (France Telecom) †
38–40 rue du general Leclerc
92794 Issy Moulineaux Cedex 9, FRANCE

Abstract

We introduce metaprogramming in a completely informal way, and sketch out a theory of it. We explain why it is a major stake for computing today, by considering the processes underlying software development. We show, from the same perspective, how metaprogramming is related to another challenge of computing, the free availability of the sources of software, and how these two phenomena naturally complement each other.

1 Introduction

In the primitive settings of early computing, the size of machines was so small that a one man's mind could fully embrace the whole of a program's activity down to the least details. Now, the human faculties haven't evolved since, whereas the capacities of computers have increased at a sustained geometric progression. To benefit from the technological advance, ever more abstract conceptual tools and programming languages had to be developed and used.

But however great the progress made by programmers as individuals, the level of complexity has been long exceeded beyond which no one can conceive the entirety of a program that would take full advantage of existing computers. It is therefore of utmost importance to develop methods of exchange, cooperation, accumulation, construction, so as to elaborate complex programs; such is the field of software engineering [Brooks, 1995].

Now, central to any process of building software is manipulation of the source code of programs. To enhance these processes, to ease the task of the programmer, is to relieve the programmer from all repetitive and conceptually redundant operations, so that he may focus on the essence of programming, that is, on the problems that have never been solved yet. This means manipulation of code must be

*This is an admittedly poor translation of the article "Métaprogrammation et libre disponibilité des sources, deux défis informatiques d'aujourd'hui", published at the conference "Autour du Libre 1999". It is freely available under the GNU GPL, version 2 or later.

†The opinions expressed in this essay are the author's own, and should not be taken as an official position of the CNET or of France Telecom.

automated as much as possible, by having machines handle reliably all the inferior tasks that do not pose theoretical problems anymore. And to automate programming is by definition *metaprogramming*.

Thus, metaprogramming, the art of programming programs that read, transform, or write other programs, appears naturally in the chain of software development, where it plays an essential role, be it "only" under the form of compilers, interpreters, debuggers. However, metaprogramming is almost never consciously integrated in the processes of development, and gaining awareness of its role is precisely the first step towards progress in this domain.

Now, even if we would like to focus on the technical task, which is sufficiently hard in itself, consisting in the exploration of automation methods for the software development process, it is impossible for us to ignore the precondition necessary to the use of any such method as well as to any cooperative or incremental work: the availability of sources.

This availability is less than ever a technical problem, thanks to the advent of digital telecommunications; but it is more than ever a political problem, at a time when the near-totality of information on the planet is under control of publishing monopolies organized in powerful lobbies. The Free Software movement fights for the free availability of source code, nay, information in general; it opposes the artificial tollgates that the legal priviledges of "intellectual property" are.

Metaprogramming and free availability of sources, such are the two major challenges, intimately related, that computing has to face today, with a growing importance. Both find the same cybernetical[1] justification in the necessity to adapt the software development process to programs that are more and more elaborate, that concern a wider and wider public, where everyone can only bring a small contribution each. At least, such is our conviction, that we'll try to share with you.

2 Metaprogramming

2.1 Metaprogramming in Everyday Life

It is possible to see the impact of metaprogramming on traditional development processes, by sorting existing programs as transformational processes with inputs and outputs, and considering specifically those of these inputs and outputs that can be categorized (with some arbitrariness) as "code", as opposed to "data". We'll note $n \to p$ the sort of programs n of whose inputs are "code", and p of whose outputs are "code". Then, here are a few examples of programs whose sort range in the simplest of those rough categories.

- $0 \to 0$: doesn't take or return code.
  just any "normal", basic, non-meta program.
- $1 \to 0$: takes one piece of input code, doesn't output code.
  Interpreter: takes one program as input, and executes it (a shell, a script interpreter, a Lisp listener; the latter cases suggest that internally, efficient interpreters will actually call a compiler). Code *inspector*: extracts some data from an input program (documentation, information flow graphs, statistics on productivity, etc). Automatic *tester*: automatically verifies the behavior of an input program, or benchmarks its performance under various circumstances. Code *analyzer*: studies the behavior of the input program, searches it for failure patterns, checks its correctness with respect to some specifications or some criteria.
- $0 \to 1$: outputs code without inputting any.
  Data *precompiler*: adapts and embeds external data inside code (pictures, sounds, fonts, keyboard management map, etc); precomputes trigonometric tables (sine/cosine for FFT), or encryption/decryption tables

[1] The word "cybernetics" is built from the greek root $\kappa\upsilon\beta\varepsilon\rho\nu\acute{\eta}\tau\eta\varsigma$, which designates the pilot of a ship, and already meant figuratively *governor*, which word is an etymological derivative of it. Cybernetics [Wiener, 1957] is the science of control, that is, of decision-taking and the flow of information.

(CRC, DES), outputs optimized routines for displaying icons, generates self-extractible archives, etc. Automatic program *generator*: outputs a program that's optimized to solve a given problem from a known class of such problems.

- $1 \to 1$: takes one input program, outputs one back.
  Automatic *translator* (from one langage to another), especially, *compiler* (from a "high-level" language to a "low-level" language). *Optimizer* or *Partial Evaluator*: adapts code for better performances in its future execution environment. Secondary code *extractor*: automatically produces code for exporting routines, initialization and finalization, error handling, etc. *Instrumenter*: inserts additional code for interfacing or debugging purposes. *Compacter*: produces self-decompacting code. *Stylizer*: makes code easier (or harder) to read and comprehend. *derivator*: computes the derivative of the function computed by the input program. *Solver*: looks for a program solving a given specification.

- $2 \to 0$: takes two programs as input, doesn't output code.
  Metainterpreter: interprets the second program according to the language described by the first. *Checker*: takes a specification (an assertion) and a program (a proof), and checks that the second conforms to the first (respectively establishes it). *Differential analyzer*: compares two programs, looks for differences between them, tries to spot a case when one doesn't implement the other, or to find a shortest set of modifications to transform one program into the other, etc. *Coexecutor*: executes at the same time both programs (or some combination thereof), for instance, so as to benefit from whichever is fastest depending on the cases, or so as to have them play against each other, or simply so as to let the user takes advantage of the features of both programs. *Explainer*: takes a program and an execution of it that led to an unexpected result, and tries to give as relevant as possible an explanation of the apparent or actual failure of the program (very primitive incarnations of such programs are widespread under the name of debuggers).

- $2 \to 1$: takes two input programs, returns one program.
  Code *walker*: takes a metaprogram and a program, and walks the second program according to the instructions specified by the first one. *Metacompiler*: takes the specification of a language and a program written in that language, and outputs an implementation of the latter in a third language (for instance, assembly language). *Cooptimizer*: combines two programs into one that efficiently coexecutes them. *Aspect weaver*: takes several aspects of a same putative program, and outputs a program that implements all these aspects at a time. *Update generator*: takes an input program and a modified version of it, and automatically generates data converters from the old format to its new version, or a "patch" program that will transform the old program into the new one, or an adapter allowing users of one version to run their applications together with the other version, etc.

- $1 \to 2$: takes one program as input, returns two.
  Phase *splitter*: takes a program, splits it in two, typically into an executive, "effective" part of it, meant to run in the target environment, and an adapter, interface part between the target environment and a host environment (embedded software vs control software, remote routines vs local ones in a distributed system, specialized coprocessor jobs vs generic processor computations, user software vs automated installer, routine library vs generator of optimized programs using that library, etc.).

We could go on toward arbitrarily sophisticated examples, or simply combine the above examples. We could also refine this type system to consider, beyond the mere arity of programs, the type of their arguments and results (and so on, recursively), with criteria more precise than that of being or not "code" (with each chunk of code being associated the language in which it is written), etc. These refinements are left as an exercise to the reader.

By reading of the above example list, one can't help but notice that in every listed sort exists pieces of software that are actually used daily, at a smaller or larger scale, by programmers world-wide. Thus, metaprogramming already exists, and proves its utility every day.

However, inspection of the same list leads to one's remarking that most of these metaprograms are written in an *ad-hoc* manner, and *a posteriori*, according to immediate needs, without any general

framework to develop them (though with a disconnected collection of tools), and worse even, without any framework to think them. Certainly, metaprogramming is present, but few are those who are aware of it, and fewer are those who actively use it [Pitrat, 1990]; the tradition in computing seems more prone to create particular cases for every of the set out concepts than to put them together through a useful and systematic theory.

2.2 Outline of a Theory of Metaprogramming

Metaprogramming isn't particularly complicated idea to formalize, once you decide to do it; actually, all the underlying ideas already exist, and await to be gathered into a coherent whole. Below is shortly sketched out such a theory, accessible to anyone with a basic culture in theoretical computer science or in mathematical logic. We invite other readers to read ahead, and skip without regret the paragraphs that be too hard to them. As for those who'd be interested in more technical details rather than less, we're preparing another, more formal, article, for them.

First, we need to define the abstract notion of a computer *langage*. After examining the properties we expect from such notion, we can identify it to the notion of abstract type of data structures, or to that of classes of syntax trees, or even to that of "abstract algebra". To achieve that, we may start from the notion of *inductive structure* (or "initial" structure) freely generated by a *signature* (or abstract grammar), that is, constituted of the terms that can be written from the constructors given in the signature. To every such structure is associated a logical theory wherein the fact that the structure be freely generated corresponds to a schema of inductive reasoning (proof by induction). From freely generated structure, we can obtain new, more "constrained", structures in two ways: firstly, by adding well-formedness requirements on terms (sorting, typing, lexical rules, etc), which is the same as restricting the considered terms to only part of the original structure; and secondly, by adding equality relations between terms (for instance, commutativity or associativity of some operations), which is the same as quotienting the structure by an equivalence relation. For purposes of relevance, we make sure that the constructors and constraints that define the source language satisfy some strong enough criterion of *effectiveness*: computability, primitive recursivity, computability in polynomial time, etc, so that the synthesis, the analysis, the comparison or the copy of terms be operations effectively simple to implement.

Then, we can define the notion of a *mapping* between langages as a binary relation between the elements of a language and those of the other language. We thus introduce the notion of a *semantics*, as a mapping between a "concrete" language and another "abstract" language, that maps every "signifier" term in the concrete language with its possible "signified" terms in the abstract language. We'll be especially interested in *observational semantics* that give as an abstract signification to concrete terms the set of valid answers to a set of "relevant" observations on these terms. We'll also introduce effectiveness criteria on mappings between languages and on semantical relations; for instance, in the most general case of observational semantics, we'll require every observation to be semi-computable.

A *programming langage* is then the data of a source language and a semantics that relates this source language to a set of observable behaviors.

By describing these mappings between languages themselves with programming languages, we obtain the notion of *metalanguage*, whose terms are *metaprograms*. We'll be able to assert that some metaprograms translate faithfully a language into another if they preserve the semantics of terms (that is if some diagram commutes). It is thus possible to express various criteria of correctness for metaprograms, to begin with interpreters and compilers. With appropriate effectiveness criteria, we can thus use such a framework to express not only computations, but also logical reasoning on programs.

Finally, when a metaprogramming framework as described above is able to represent itself, it is said to be reflective. Such a framework allows not to have to introduce a new metalanguage (with all associated

axioms) each time we want to explore the behavior of the system in a deeper "meta" direction; the distinction between code and data is thus blurred, which leads to a more integrated vision of software development [2].

2.3 Multi-Aspect Programming

If we investigate existing applications of metaprogramming, we realize that it is used to automatically manage the transition between several different aspects of same computational objects: for instance, when one compiles a program, one is interested in the "same" program, under different forms (source or object code), each suited to its own set of tasks (human modification, or machine execution).

Conceptually, we consider a "same" program, a "same" object, a "same" idea, independently of the various representations through which we communicate and manipulate them; one form as well as one other can be used to describe the whole of the "interesting" properties of an object. Nevertheless, whatever the chosen representation, there must still be one, and considering the effectiveness criteria imposed by physical and economical constraints, one would better chose a representation "adapted" to implementing as efficiently as possible the manipulations expected to be effected on the considered object.

Now, the "interesting" properties of an object will inevitably vary through time and space, depending on the persons who work with the object, on the machine components that handle it, on the field of activity, the expectations, the proficiencies of ones, the usage goals and the technical constraints of others. To provide each program and each user at each moment with a representation of the considered objects that be as adapted as possible to his current interests, there needs be metaprograms that enforce consistence in time and space between the various aspects of these objects.

In this way, a plane will be for a engineer designing the fuselage, a set of curves and equations of which some parameters must be optimized, for a component manufacturer, it will be a schedule of conditions, for the builder, it will be an assembly process, for the maintenance officer, a set of tasks to perform, for the restoration engineer, a set of problems to fix, for the hardware manager, a set of spare parts, for the pilot, a ship to take to destination safe and sound, for the passenger, a disagreement to minimize to get to another place, for the traffic controller, a dot to route among many others, for the commercial clerk, a set of seats to fill, for the commercial strategist, an element of a float to deploy, for the director of human resource, people to manage, for the accountant, a historic of transactions, for the insurance agent, a risk to evaluate, etc, etc. The computerization of some or the whole of the management of the life of a airplane implies providing each actor in presence with a point of view of the plane that suits his needs, that will contain parameters most of which are only useful to him, but some of which also involve other actors, with whom it is essential that some agreements be found, that data be synchronized. Metaprogramming enables to handle this synchronization, this consistency between the points of views of multiple actors [Kiczales *et al.*, 1997].

Even for the computing projects that are simpler than the complete management of a air float, there necessarily appears a divergence of point of views, as soon as multiple actors (programmers, users) or in presence, and/or as soon as these actors evolve and that their points of focus change with time (a same person can take several roles in succession). As soon as a problem is complex enough, no one can embrace at once all the aspects of it, and it is necessary to treat them separately. Hence, metaprogramming is useful to enforce consistency between these aspects, that else would have to be managed manually; it allows to handle once and for all an uninteresting aspect, so as to forget it afterwards.

[2] The theory sketched out above allows for formal modelling of metaprogramming and of the semantics of metaprograms; however, it doesn't suffice to capture the whole phenomenon, since the choice of which structures to consider as "code" instead of "data" is mostly arbitrary in the above presentation. Actually, it is entirely a matter of point of view, and it requires to be an observer exterior to this modelling so as to have a criterion suitable to "objectively" distinguish "code" from "data". We'll see (section 3.2) how this distinction can dynamically emerge as a by-product of the development process.

3 Expressiveness of Metaprogramming

3.1 Expressiveness and Computability

We have presented metaprogramming as a very useful technique (and a very used one, despite lack of awareness of it) to solve some problems. The question is thus to determine whether this technique be original, or would only be a combination of well-known existing techniques (which combination it would then be interesting to detail), whether or not it brings new solutions to known or previously unsolved problems, whether, in the end, it be an essential and indispensible technique, or on the contrary it would be but accessory.

This question is actually that of the expressiveness of programming languages, as algebras allowing to combine multiple techniques: what makes a language more expressive than another? what enables the programmer better or worse to solve his problems? And, to begin with, what are these problems, how can they and their solutions be characterized, compared?

The fundamental result concerning the expressive power of computation systems is the one by Turing [Turing, 1936], that shows the existence of a class of functions, said to be computable, able to perform any thinkable mechanical computation. Given a set of possible inputs (questions) and a set of possible outputs (answers), both digitizable (encodable with, for instance, natural integers), any mechanically implementable programming language can express at most as many functions from the input set to the output set as a Turing machine. A programming language that can express all computable functions from one set to the other is said to be universal, or Turing-equivalent (assuming both sets are infinite). All such languages are as powerful as all others, from the point of view of the functions they can express.

Nevertheless, it is obvious that not all Turing-equivalent languages are worth the same from the point of view of the programmer: all those who made the respective experiences will agree that it is easier to program in a high-level language (like LISP) than in a low-level language (like C), which in turn is easier to use than assembly language, which is better than binary code, which is simpler than the transistor-per-transistor design of a dedicated electronic circuit, or the specification of a Turing machine. Actually, Turing's result is but the very beginning of a theory of the expressive power of computing systems, and certainly not the end of it. To stop with this mere result, and say that "since all languages, in practice, are not equivalent to each other as they are according to Turing, then theory cannot say anything", would be to abdicate one's reason and to look in a vague nowhere for an unspeakable explanation, it would be giving superstition and ignorance a gratuitous and preposterous dignity.

Before we go further, let us remark that the very demonstration of Turing's result is based on metaprogramming: universal languages are equivalent one to the other in as much as it is always possible to write in one an interpreter for any other one, so that any program in the other language can find, through the use of a translator, an equivalent in the starting language. Besides, this is precisely the reason why these languages are called *universal*. Now, metaprogramming is a programming style ignored by all the software development methods as applied to most languages[3], for it consists precisely in not using the studied language, but another language, through metaprograms. We may thus wonder how can this reluctance can be formalized, and what will then remain of the equivalence between universal languages.

3.2 The Process Matters

There unhappily, doesn't exist any fully satisfying theory of expressiveness yet; the only recent work that we could find on the subject [Felleisen, 1991], very interesting as it is, limits its ambition to the relative macro-expressibility of extensions of a same language one into the other. However, we do have

[3] However, see the case of languages of the FORTH and LISP families, as well as in a limited way, that of "statically typed" functional languages.

some scattered elements of a general theory, that we think are largely enough first to justify the rather "intuitive" idea that metaprogramming brings an increase of expressiveness, and then to refine or reject a few common assertions heard about the "excessive" expressiveness of some languages.

The key point on which the notion of computability fails to express(!) the expressive power of programming languages, is that software development cannot be summed up as the writing of a one perfect monolithic program that either succeeds or fails to solve a given static problem, but is instead a dynamic and evolving process, in which man and machine interact in a more or less elaborate way.

Thus, for any given program to write on a given machine, the truly optimal solution can only be expressed in binary code, and contains lots of nifty "hacks", of encodings based on fortunate coincidences, of "puns" on the encoding of data, addresses, and machine instructions. But the fact is that finding such a solution would demand gigantic efforts, and the advent of new hardware architectures will make all these pun optimization obsolete. Now the human forces spent during the development are important; there are even essential, since the ultimate purpose of a computer, like that of any tool, is to minimize the amount of human efforts required for the completion of ever more advanced tasks. The relative cost of human and mechanical resources may have been such that, in the 50's, the hand-writing of super-optimized binary code was economically feasible (see the story of Mel, exerpt from the Jargon File [Raymond, 1996]); such thing is impossible today.

Thus, the problem is not only technical, it is also economical, moral, and political, in as much as it involves shifting of human efforts. In the development of computer programs, as anywhere else, *the process matters.* And it is this very process that the popular notions of code reuse, modularity, dynamic or incremental programming, development methods, etc, attempt to enhance, though without a formal rational approach. Similarly, the trend in computer science to leave too low-level languages in favor of higher-level languages is precisely due to human intelligence being a limited resource, a scarce one indeed, that has to be saved for the most important tasks, the ones where it is indispensible (if not forever, at least for now).

Therefore, a satisfying modelling of the expressiveness of programming languages, even though it be abstract enough not to overly depend on ephemeral technological concerns, must take into account the man-computer interaction in a way that includes a notion of human cost (and perhaps also a notions of error and confidence). We suspect such a modelling to be possible using the theory of games.

3.3 Complexity of Incremental Programming

Though we lack a consistent algebraic theorization of expressiveness, at least we can give the following first intuitive approximation , taking into account a dead simple notion of human cost: a programming system will be more adapted than another to a given usage pattern if it requires "on the long run" less human interaction to solve in succession an indefinite series of problems within the given domain. The usual approximate "metrics" of production can be used (number of lines of code, or abstract syntactic constructs, or of symbols having been typed, number of man-months spent). This approximation provides only a gross characterization of expressiveness, and doesn't allow to define an efficient programming style, but suffices to justify use of metaprogramming.

In the traditional development process, that excludes metaprogramming, the near totality of the code constituting software is hand-typed by humans; the details of execution, the overall behavior, everything must follow directly from human thought; every stage of the development, every modification is the work of man. Now, the catch is that sooner or later, some new functionalities to add to a software project will require global architectural changes of some depth: all the global modifications of the program will have to be performed by hand, with all the inconsistency problems unwillingly introduced by these changes, that induce numerous bugs and cost lots of iterations of development (for a publicly documented

such phenomenon, see the occasional API changes in the Linux kernel; for a spectacular example, see the explosive bug of the first Ariane V rocket). The alternative to these modifications is a complete reimplementation, whose cost is that of a rewrite from scratch. All in all, the incremental cost of traditional development is proportional to the size of code that differs between two iterations of the process.

Let us now make metaprogramming part of the development cycle. Since metaprogramming allows arbitrary processing of code by the computers themselves, it enables the programmer who has to face a structural change in his program to perform semi-automatically any task of instrumentation or transformation of his program, so as to make it conform the new architecture, to check his new code with respect to declared invariants, to enforce consistency of various parts of his program. "Semi-automatically" here means that the simplest "metaprogram" to handle a few particular cases sometimes happens to consist in manually specifying them case-by-case. In brief, the incremental cost of software development between successive iterations of process is proportional not to the size of their difference in code, but to the size of the smallest metaprogram that allows to transform the previous state of the project into the new one, that is, to the conditional Kolmogorov complexity [Li and Vitanyi, 1998] of the next step relatively to the previous one!

In the worst case, metaprogramming won't have served, and the cost will be exactly the same (up to a negligible additive constant) as that of traditional programming[4]; theory tells us that, by the very definition of the concept of randomness, this means that the project has evolved in a completely random way with respect to the existing code base. In the best case, there can be arbitrary gains. In general, metaprogramming allows for optimal cost with respect to complexity; its gains as compared to the traditional approach are proportional to the size of the project in the inevitable case where structural changes with global effects are necessary.

Thus, the theory of Kolmogorov complexity allows us to see how metaprogramming always win, and most often quite a lot, over non-meta programming, by optimizing the sharing of information between iterations of the development process. Besides, it is clear how the tiny doses of metaprogramming used in traditional software projects are a determining factor for the continuation of these projects: choice, often static, of a development environment; filtering of programs with static analyzers, checkers of more or less sophisticated invariants, or other kinds of `lint`; use of editors that manage the indentation and structure of programs, that they can colorize and fontify; occasional use of search-and-replace tools on the source code; in the case of more daring hackers, "macros" and "scripts" written in elisp, perl or other languages, to edit and manipulate sources.

However, and this will prove very important in a further part of this essay, metaprogramming wins if and only if on the one hand an infrastructural effort is spent to allow easy manipulation of programs, and only if on the other hand there is enough redundancy between successive developments to allow some non-trivial sharing of information accross time; the benefit enabled by metaprogramming is thus asymptotic: it is low to begin with, but gets better quickly with the size of programs and the number of development iterations, that, with the space-time *extension* of software development projects.

[4] In any case, metaprogramming being an extension to traditional programming, and offering a choice of additional tools without removing any prior option, it can but do at least as well as traditional programming, in terms of the minimum amount of work required to complete a task.

4 Availability of Sources

4.1 Metaprogramming vs Intellectual Property

Let us admit the interest of of metaprogramming techniques. Why are they not more commonly known and consciously used? Just what is it that has held up their spreading more widely?

We personally find it obvious that barriers to the distribution of sources are as many brakes to the development of metaprogramming. In fact, the very condition for the use of a program-reading metaprogram is the availability of a program to read; the condition of usefulness of a program-writing metaprogram is that the output program may be distributed and used; and these conditions combine when a metaprogram at the same time reads and writes programs, and even more when the metaprogram depends on the long term accumulation of knowledge about programs! Every limitation on the manipulation of programs is as much of a limitation on the feasability or the utility of metaprograms, and a discourages as much their potential authors.

Now what are the implications of the current regime of "intellectual property" on software?

Firstly, there is the so called "authorship" right, which is actually a publisher's right, since every author abandons all his rights to the corporation that salaries or publishes him (unless he becomes his own publisher; but then, it will be not as an author, but as a publisher, that he will earn money, which doesn't remove anything to the problem). To respect this "right", a metaprogram must refuse to make some inferences, some manipulations, that would violate (according to the local country's legislation) the license under which the input program is available. Which manipulations exactly are forbidden isn't clear.

For instance, some licenses are per-user, which forbids to a metaprogram any inference that will be useful to two users (or to a non-authorized user) for unclear and varying notions of "useful" and "user" (what happens when two people collaborate on a same document? or when one supports the another one or serves him as a secretary? what if the final result of the computation is broadcast to millions of persons?). When they are per-machine, or per-simultaneous-user, the problem gets hairier, as the notions invoked by these licenses lose any effectiveness in the presence of time-shared machines, of multiprocessors, of clusters of machines, or worse, of faster machines: no software parameter can correspond to these notions, and even less with software architecture emulators (e.g. 680x0 or x86 emulators), and worse even if these emulators use techniques of analysis and translation of code (themselves prohibited by many licenses) that completely set apart any precise correspondance between on the one hand the "abstract" machine for which the code is written and that the license expects, and on the other hand the "concrete" machine on which the program runs. The notion of user is also blurred by metaprograms that manipulate more than one program at once, and will become even more so with the hoped-for advent of "artificial intelligence". All these problems can be avoided at the enormous cost of never using as input of metaprograms but software and information that are free of rights (aka free software and free information), which, in a strictly legal setting, would require anyone who introduces any original information to sign a form (or check a button in a menu?).

A trickier problem, that isn't solved by this measure, is distribution of the results of metaprograms. For, even with free software licenses, that can be mutually incompatible, and even more with proprietary licenses, some operations are allowed, but only under condition of non-distribution of results ("private use"). But what exactly mustn't be transmitted, when a metaprogram included in its database a synthesis of the analyzes of numerous programs? Where does redistribution start? Does it cover exchange between people in a same company, or a same moral entity? What if this entity is an association that admits anyone as a member? Does it apply to machines? What if this machine is a cluster covering several ones? What if this cluster covers the entire world?

The most refined is the existence of patents. These affect metaprograms universally, independently from the base material being used and their licenses. Patents do not introduce any structural rules that forbid a logical inference in a metaprogram; rather, if at a given moment, the metaprogram entails the use of some algorithms for some goals, then the inference that induced this result must be retroactively invalidated, least some license be obtained from the patent holder. But since the notions of algorithm and goal are quite fuzzy, it would require quite an elaborate meta-meta-program to monitor that the metaprogram never infringes any of the millions of registered patents.

If all the above rules concerning the use of metaprograms seem ridiculously absurd, do not forget that after all, though "artificial intelligences" differ greatly in their current quality from "natural intelligences", they are based on the same principles, and the same constraints. Hence, be aware that the most widespread metaprogram in the world, that is the human mind, is the first to be compelled to these absurd constraints, by the same absurd laws!

One can understand, under those conditions, that only the most simplest metaprograms could be usable and used enough to be actually developped. No deserving artificial intelligence is to be expected, and no universal database, with every cooperative effort being constantly impeached by the barriers of "intellectual property" [5].

Writing and using metaprograms that abide by current laws is a hell. There is no logically consistent way to define rules for or against the use of metaprograms, even the simplest ones, in presence of "intellectual property". Thus, the only law is the rule of the strongest, the one whose millions may bribe the best lawyers to defend their "rights". We think that it is not possible to define any notion of "intellectual property" whatsoever that be technically enforceable in a legal frameset, and that be not deeply unjust and economically harmful; in another document [Rideau, 1997], we have given a thorough argumentation in favor of this point of view. However, if such a notion did exist, the burden is up to the proponents of this "intellectual property" to expose a definition of it, and to prove with any possible doubt its beneficial character.

Actually, as long as the only operation that leads to production of code is manual addition, the writing out of the blue by a human mind supposedly inspired directly by the Muses, then it is possible to attribute an "origin", an "author" to every word, to every symbol that constitutes a program. Now, as soon as is

[5] Once again, the arguments that support liberty to use and distribute information are of the same vein as those that support liberty of trade [Bastiat, 1850]. The barriers of "intellectual property" are worth the barriers of tariffs and fees at the borders of countries, at the entrance or the exit of towns, at the crossing of roads and bridges; both kinds of barriers oppose the free trade of goods and services; both kinds of barriers are as certainly artificial that their enforcement requires the use of armed force against citizens who willingly enter exchanges that do not deprive the resources of any third party.

Creating, researching, distributing, teaching, correcting, guaranteeing information are as many services that men are naturally inclined to exchange against other services. The barriers of "intellectual property" are as many obstacles to the exchange of these services, and establish the monopoly of a publisher on all the services relating every piece of "proprietized" information. The market of information services is thus divided up more than was the market of goods during the middle ages, for the barriers are now around every person, every company, every institution, every publisher, rather than only around towns, roads, bridges and countries. The consumer and the entrepreneur are the permanent victims of this state of fact that law unhappily supports.

This protectionism in the field of information services, as harmful as it is to the industry, isn't in any way justifiable by a natural right to intellectual property. In fact, if goods, as physical resources limited in extension, are naturally prone to be controlled and possessed, information is intrinsically and indefinitely replicable and shareable, and doesn't lend itself to the least control, to the least possession. Besides, this is the very reason why some request *the law* to establish an artificial "intellectual property", whereas material property by far preceeds law, that only confirms it. We can very well see that those requesters, like all protectionists do, only ask for legal plunder, which costs twice to the whole set of consumer-citizens what it yields to their own personal or corporate benefit of producers.

Finally, the free movement of information is necessary to define equitable terms in exchanges. It is thus necessary to the correct balance of economical markets, et any attempt to this free trade biases markets, induces financial bubbles, and brings ruin.

allowed metaprogramming, that is, arbitrary operations that act on code and produce code, as soon as is considered the environment inside which lives the author (man or machine) of a program then this one is no more the inventor of the program, but only the last link of a holistic process of transformation, in which it is not possible to attribute an unequivocal origin to any produced element whatsover.

4.2 Partitioning and Loss of Responsibility

Beyond the previous technical and legal considerations, the whole system of proprietary software development is completely opposed, in its essence and its existence, to metaprogramming.

In the traditional closed-source development process, software is written in a partitioned way, within the barriers constituted by proprietary licenses. Every developer is only passing by during the development of a piece of software; he has no control over its destiny and isn't concerned with it; within a few years, the software is doomed to end in an attic, and he himself is doomed to end out of the owning company, by pure marketing decisions, the first victims being users. Those users, having no access, direct or indirect, to the sources, are completely helpless. Every group of developers has access only to a small part of the whole that constitutes the software system as it is meant to run, and, which interests us most, as it runs during development. Thus, developers have no access to the infrastructure above which they write their programs; should this infrastructure prove insufficient, they have to accept its limitations and work around those they can cope with, or they must rewrite all of it from scratch, unless they find a way to have it modified by its authors.

Writing or rewriting of development tools is very expensive in terms of material and intellectual resources of development; in the case where the rewritten infrastructure itself remains proprietary, to comply with the surrounding development model, it leads to reinforce the partitioning of developers, by introducing incompatible development tools to which every new collaborator must be trained. Moreover, these tools ar often imperfectly designed, by people for whom such is not the primary calling, who do not have adequate training or culture, who often ignore the state of the art of scientifical research on the topic, and who don't have the resources necessary to track it. In the world of proprietary software, inventing a new language or tool is thus a nearly impossible and excessively costly wager. Unless he has the means to be master of a large captive market, so as to be able to impose the direction to be followed by the evolution of "meta" products, it will be in the interest of the inventor of a tool with a universal or however general vocation, *as a user* of this metaprogram, to break the logic of partitioning and to freely distribute it[6].

Another possibility open to unsatisfied users of some proprietary infrastructure consists in bending the position of the providers of the infrastructure; now, there is no solution to this problem that will be reliable for long-lasting, non-superficial effects, unless these providers themselves be bought, at the cost of millions of dollars and administrative reorganizations. Lastly, there remains to those users the possibility to participate to standardization commitees, that allow to some extent for feedback from users to providers, despite administrative time-wasting and political arguing. But within the framework of normalizing software whose major providers follow the proprietary model, the wishes of users are all the less important than the capitalistic and technological weight is concentrated in the hands of providers, who thus possess most of the decision power. Besides, no decision can be taken without the providers already having elements to implement it, and consequently, not without their having already taken the major decisions regarding the evolution of their products. Admittedly, the providers will make their products evolve so as to satisfy their customers, but between the real needs of users and the final decision

[6] There is at least one commercial project that, being firmly oriented in a metaprogramming process, followed this path: the project for the development of distributed mobile systems from Ericsson, for which was specifically developed the langage and infrastructure Erlang/OTP, now published as free software `http://www.erlang.org/`.

are enough middle men and unfaithful interpreters so that these needs have only a marginal importance, in the end.

The benefit of metaprogramming, as we saw, resides in the automatic management of redundances in the code, through space and time. Partitioning, by restricting the space-time sharing of program sources, prevents programmers from confronting the code resulting from their respective experiences, reduces the opportunity to detect redundances, to simplify code, to automate propagation of information; it holds back or annihilates the benefit of metaprogramming, and encourages more low-level programming, writing of throwaway programs, without either past or future.

What are the consequences of this partitioning on the code itself? The developers of various components are made irresponsible with respect to each other, and particularly with respect to various "meta" levels. They never engage in a metaprogramming approach where they would develop tools powerful and generic enough, for, being isolated, they would have to pay the whole cost of it while only benefitting from a tiny part of it. Thus, they interact through crude interfaces (API), stacked into similarly crude hermetic "layers". These API, so as to be commercialized, must be based on standards independent from these proprietary development tools, hence, are built, by a levelling down, on extremely precarious "smallest common denominators", devoid of any abstraction power (libraries interfaced to C or C++; bytestreams; CORBA); most of the semantics of these API cannot be expressed in the computerized language of the interface, and must be explained informally in the documentation; any verification on the correct usage of these API must thus be done by hand, which makes undetected mistakes and inconsistencies very frequent. Developers not having control on the languages they use or their implementation, these languages do evolve little or badly, being guided by vague "benchmarks", that by thermostatic effect, encourage programmers to adapt more and more to "acknowledged" programming styles, independently from their technical qualities or lack thereof. Providers of proprietary tools, once they have reached the "standard" of the market, extend their tools so as to cover a large range of fringe needs that be spectacular enough to impress non-technical managers, but most often leave behind (but out of luck) the essential features that would fulfill the deep needs of developers.

The bitter fruit of industry adopting this paradigm of partitioning is thus a culture of developers by made irresponsible. Programmers, without a way to intervene, learn to accept without a critical mind, without even the least notion of appreciation, even the most basic one, the development tools and the programming languages that are imposed to them in the end by a management that has no proficiency whatsoever to judge their worth. Any "market decision" concerning programming languages henceforth confirms itself, independently from any direct technical criterion, influenced principally by the inertia of what exists, and by the intellectual bombarding by marketing services of providers. In this way could be established languages like C++ or Java, that are resolutely inferior to the state of the art regarding programming languages.

In this way also are established, by self-stabilizing effect, monopolies on every segment of the market, with the most aggressive monopoly eating the other ones, and preventing any progress from happening without it. Certainly, within these monopolies, the sharing of resources, if it happens, can generate interest for metaprogramming; but the absence of competition doesn't encourage using it, and neither does the state of general lack of responsibility among computer engineers that are recruited from the outside world, nor does the necessity for any communication with this world outside the monopoly to use closed opaque formats, designed to prevent metaprogramming. Even though every monopoly can benefit to some extent from metaprogramming techniques internally, partitioned development still has the result of a loss of interest for metaprogramming, and considerable physical and moral damage for the industry at large.

4.3 Open Development Process

If the paradigm of partitioned development of proprietary software entails a loss of interest for metaprogramming and the very corruption of development tools, then *e contrario*, the advent of free computing, wherein software are developped in an open way, with complete availability of sources, will bring along more responsible programmers and development tools of better quality. These sole effects justify the growing success of the free development model, as currently observed.

In an open development process [Raymond, 1997], no barrier, no arbitrary limitation, no irresponsible diktat from management; quality matters above all. For the developer community will naturally [7] carry its efforts on the most adapted code bases, that will be most easy to extend and maintain. Developers are made responsible for the software they produce, and for the software they use.

Hence, and even though few are fully conscious of it, this phenomenon appears also at the meta level, by the choice and the development of free development tools. It isn't out of luck that the world of free software has generated so many new development tools that users do not hesitate to extend, beyond any existing standard but in an open way, thereby verily creating new standards: thus the languages and/or the implementations Emacs LISP, GNU C, Perl, Python, GUILE (and many other Scheme implementations), Tcl, bash, zsh, to only name the most popular ones, as well as languages that came out of academia like Objective CAML, Haskell, Mercury, etc; but also, let's not forget them, tools like (X)Emacs, GNU make, CVS, diff, patch, rsync, and actually, everything that constitutes free Unix systems (GNU/Linux, BSD), and together with them the infrastructure of the Internet.

The users take an active, nay interactive part in the elaboration of these tools and their implementations: they suggest enhancements or extensions, denounce errors, and if necessary, they will themselves implement these enhancements or extensions, and correct bugs. This is the ground level of metaprogramming, yet it is completely absent from the proprietary software development model. There remains to see whether free software will tend to confine metaprogramming to that level, or whether it will bring it to levels never reached before.

Now, since the study of the development process strongly tells in favor of metaprogramming for reasons of complexity, and that metaprogramming and free availability of sources seem to be intimately related, let us see if a simple argument on this development process wouldn't justify both metaprogramming and open development at the same time. Now, in the same way that metaprogramming was being justified by the optimization of information sharing within the development process throughout its timely extension, open development finds its justification in the optimization of information sharing within the development process throughout its spatial extension. From the moment we consider this process in its space-time extension on, and knowing that space and time are ultimately convertible one in the other, it becomes clear that both considerations are akin and can be merged into one: that a programmer communicate with a future self, or with another programmer, there remains the problem or control, of decision, in a world where many actors interact. It can be thus understood that the coming of the Internet, which multiplies the speed and bandwidth with which potential developers exchange messages, amplifies the utility, the necessity even, of methods to efficiently handle shared information, that is, of metaprogramming and open development[8].

So not only metaprogramming and free availability of sources are related, in the sense that one enables benefitting from the other in a way not otherwise possible, but this relation is a strong and structural one: metaprogramming and open development are actually two faces of a same phenomenon that unifies

[7] "Naturally", that is if the community is well-informed enough, and doesn't suffer too much from the repercutions of the marketing hype and of the displacement of capitals due to the surrounding development model.

[8] Thus, the Internet is both the vector and the result of a world-wide cooperation of hackers around freely available sources.

both of them: reflective programming (which naming we are going to justify), that optimizes sharing of information within the development process through space and time alike.

5 Reflective Programming

5.1 Reflective Systems

Let us examine the properties of a reflectivee development framework, that would integrate both metaprogramming and free availability of sources. Such a framework would contain all the tools necessary to programming and metaprogramming, and the sources of said tools would not only be freely available in their whole entirety, but would be designed so as to be semi-automatically managed, by the system, under the guidance of the developers. The word reflective is justified like this, because the system tools somehow manipulate "themselves"[9].

The free Unix systems (GNU/Linux, *BSD, etc) as well as some others (like Native Oberon) are as many complete reflective systems: they are freely available, and come with the sources of all the software tools required to their self-development, from the user interface to the compiler and to the drivers for peripheral devices. But their "reflective loop" is very long, which identifies programmer and metaprogrammer. If we consider as "given" the operating system and its basic development tools (including a C compiler), then, the many programming language implementations written in the implemented language itself are as "reflective", and with a shorter reflective loop between implementation and usage; however, these implementations do not constitute complete systems (and do not usually seek to become so), since they depend for their development on numerous external services.

Now, these reflective systems, whether they be complete or not, make little use of their reflective capabilities, that remains essentially a matter of implementation: the sources neither are designed to be manipulated by anything else but compilers or interpreters of the languages being used, nor are they produced by anything else than a text editor. Uses of reflection for purposes of analysis or synthesis to this date are very limited and decoupled one from the others, and they do not involve any persistent or accumulative process. Moreover, the languages being used are not prone to use by metaprograms; they are designed to run in existing closed systems, and permeated with a culture of partitioned development, that excludes any metaprogramming; sometimes, they date from the heroic times when metaprogramming what extremely limited by the feeble computational and memorial capacities of computers.

There henceforth still remains to develop an authentically reflective platform, where metaprogramming in all its forms would be integrated to an open development process. There exists, as far as we know, only one project resolutely oriented towards this direction[10] but it is still in an early phase, for it lacks what would no doubt constitute the heart of the system: a programming language adapted to a reflective style of development.

5.2 Reflective Style

What will computing systems of the future be like, reflective as they will be? What reflective programming style will suit software development within such frameworks? How will the art of programming be affected by the growing impact of metaprogramming, used on a large scale on freed sources?

Turing-equivalence ensures that whatever be the language initially chosen, metaprogramming will allow the renewed development of tools as adapted as possible to the domains being explored by the

[9] To avoid paradoxes, this assertion requires a few precautions so as to be rigorously formalized, which precautions are beyond the purpose of this essay; we are preparing a technical report dedicated to this exercise.

[10] Project *Tunes*, `http://www.tunes.org`

programmers: this is the problem of "bootstrap", well known from all implementors of compilers written in their own target language. In brief, it ensures that an open development process will eventually lead to metaprogramming, and, on the long run, to a reflective system. But it doesn't give any indication as to what a mature reflective system will be like.

Such an indication lies probably in the experience of existing dynamic development systems, like languages of the FORTH or LISP families (or their crossing, the RPL), that possess an almost instantaneous edit-compile-run-debug interaction loop, and have integrated various original techniques for metaprogramming in their daily practice: internal evaluation, escapes allowing for "immediate" execution of code during compilation, arbitrarily powerful "macrodefinitions", "meta-object protocols", and so on. These techniques have been used, among others, to implement internally to these languages, sometimes with extreme sophistication, various programming paradigms (object systems, logic programming, concurrent evaluation, etc), language extensions, domain-specific languages [Shivers, 1996], and so on. The systems built around these languages suggest that a reflective system would be "integrated", with all its functionalities being reachable from a same language, in a way that minimizes the redundance of linguistic efforts between meta-levels, and that justifies the naming "reflective" with respect to the remarks in section 2.2.

These languages are no doubt by this virtue the most expressive languages currently available, and we conjecture that if used with this point of view, they are those that can bring the highest productivity. However, we can but note that they haven't "broken through" overly within the industry, though they have lasted a remarkably long time, and have technically seduced those who have tried them. In addition to the general reasons already conjured about the effects of partitioned development on the industry, it seems to us that, paradoxically, a reflective style is a handicap rather than an advantage for languages provided within the partitioned proprietary software model: a tight integration to closed development software has too heavy costs, both direct and indirect, to the user, first in the form of limitations in deployment (restricted choice of hardware, licenses, royalties), but above all in the reliance of important technical and human investments upon a technology whose evolution the user doesn't master, since it requires the good will of the only the proprietor of the development tool to have it evolve or even just survive. Only the free availability of sources provides the ability to adapt and deploy, and guarantees the perenniality necessary to investment in an integrated software solution.

Another path leading to comprehending what a reflective programming style will be like probably starts among existing formal methods in computer science: theoretical frameworks for programming, techniques for proving properties of programs, techniques for synthesis or extraction of programs, type systems allowing to establish interfaces between modules of a system. In fact, any non-trivial metaprogram must be able to manipulate programs no simply according to their superficial shape, their syntax, but also to their fundamental meaning, their *semantics*. Only by mastering the semantics of programs, by revealing their essential structures and eliminating superfluous complexities may it be possible to build, analyse, communicate programs, and to negociate upon solid bases contracts between requesters and providers of services[11] to interact and cooperate in an efficient and responsible way. Besides, let us remark that these actors of the computing world, requesters and providers of services, may be humans

[11] There is a proverb too commonly repeated, impressed with folly more than with wisdom, according to which the expressiveness of a language would reside not in what it allows to say, but in what it forbids to say. By following this logic, the most expressive language would be the one that would forbid to say anything! Our study suggests that the expressiveness of a language would rather reside in the contracts that it allows to conclude, which contracts can include positive as well as negative clauses. At the opposite of "fascist" recommendations of fanatics of bondage and discipline, or of the dubious slogan of "information hiding" repeated by big shots of "object oriented" closed development, the goal is not to hide or forbid information, but to communicate and use metainformation relating to the use of other information; this metainformation, impossible to formally express in the traditional frameworks of closed non-meta programming, is nevertheless necessary to negociate adequate contracts between computerized of computer-using actors.

as well as computers, or as an integration of both; there remains between them the need to communicate meaning, and not random sequences of zeros and ones.

Meaning, here like elsewhere, emerges as an epiphenomenon of interactions between communicating actors, of their exploration, superficial or deep, static or dynamic, actual or putative, of the communicated semantical structures. Whence again the necessity of the free availability of sources, so that the meaning of these structures be fully accessible. Hiding the sources of programs is putting a brake on the communication of described structures; and partitioned development indeed leads to using low-level structures, that are both semantically poor, and ridden with useless cumbersome details. A reflective programming style will on the contrary seek, dynamically if needed, to make explicit as much as possible the meaning of exchanged concepts, while eliminating as much as possible the useless details.

5.3 An Industrial and Scientifical Necessity

We dared speak above about the relative productivity offered by different languages. Now, there doesn't exist as far as we know any serious scientifical study currently possible on the subject; the unit used to meter productivity is generally the thousand of lines of code, whose meaning in terms of functionality greatly varies from a language to another, and depending of the considered problem domain. To manage serious and unquestionable scientifical studies would require to have a large base of code that be visible from the whole scientific community, that would be possible to inspect in detail and to evaluate considering provided functionalities. But the very principle of closed development systems that dominate the whole of the industry is that code be not publicly accessible of evaluation, scientific or else. Peer evaluation on the other hand is the very essence of open development [Raymond, 1997], which is the only method suited to having scientific studies go forward. It is also obvious that to analyse scientifically a growing quantity of information, it will be necessary to use metaprograms more advanced than mere counters of lines of codes, of lexems, or of any other syntactic unit. Reflective methods are thus a scientifical necessity.

They are also an industrial necessity. Closed non-meta development makes the industry lag more and more behind research, for it constantly commits itself blindly into dead-ends with all its inertia, and cuts itself from any rational control (see partitioning and loss of responsibility above). Only scientifical evaluation can guarantee to any industrial entrepreneur, or to anyone, that is, to any consumer, that his interest is taken into account by computer service providers, be them internal or external developers. We have a glaring illustration of that, at a time when everyone gets mad about the infamous "year 2000 bug", which currently necessitates the correction or replacement *by hand* of millions of lines of code, whose source is sometimes lost! How many efforts would have been saved, should reflective methods have been used during development of that code! Not only would the code, freely available thus shared, would have been shorter, hence simpler to replace, but its visibility would have greatly eased detecting occurences of the bug, and correcting them promptly, instead of hasarduously working around them; lastly, metaprogramming and the use of high-level languages that it recommends, would have allowed to automate most of the process of global conversion of programs and data, from simple and localized indications on the modifications to bring to the data processing.

Metaprogramming and free availability of sources consist in opening programs to all processing and modifications, made by man or by the machine; all in all, it's a matter of opening programs to intelligibility, of participating in a process of building a shared knowledge, that is, of science!

6 Conclusion

We think we have shown, with the help of cybernetical arguments, that compupting today has to face two challenges apparently different but nonetheless intimately linked, metaprogramming and the free

availability of source code of software.

Their joint advent is ineluctable, for they are the key to an important reduction of costs of production, that in the end are sustained by the consumer. To accompany this advent in good conditions, to make the best out of it and avoid its pitfalls, it is useful to know these two phenomena.

We do not claim to have made an exhaustive exposition of the subject; we only hope we have revealed an aspect of it that is usually neglected, and make the wish that software engineering be further studied under the light of a theory of complexity and expressiveness that would take into account the few elements that we have proposed.

Let us then hope that practitionners of computing arts shall free themselves from the flashy but empty slogans like "multimedia", "object-oriented", "design-pattern", "intelligent network", etc, that are by-products of a culture of partitioning and loss of responsibility. Shall they rather adopt a scientific attitude, that is open and critical at the same time, with respect to the essential process that underlie their activity, which processes, for technical that they be, nonetheless have economical, political, and all in all, moral implications[12].

References

[Bastiat, 1850] Frédéric Bastiat. Œuvres, 1850. `http://bastiat.org/`. 5

[Brooks, 1995] Frederick P. Brooks, Jr. *The Mythical Man-Month, Twentieth Anniversary Edition.* Addison-Wesley, 1995. 1

[Felleisen, 1991] Matthias Felleisen. On the expressive power of programming languages. Technical report, Rice University, 1991. 3.2

[Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. June 1997. 2.3

[Li and Vitanyi, 1998] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications, 2nd Edition.* Springer-Verlag, 1998. 3.3

[Pitrat, 1990] Jacques Pitrat. *Métaconnaissance, futur de l'intelligence artificielle.* Hermès, 1990. 2.1

[Raymond, 1996] Eric S. Raymond, Editor. The on-line hacker Jargon File, version 4.0.0, 1996. `ftp://ftp.gnu.org/pub/gnu/jarg*`. 3.2

[Raymond, 1997] Eric S. Raymond. The Cathedral and the Bazaar, 1997. `http://catb.org/~esr/writings/cathedral.html`. 4.3, 5.3

[Rideau, 1997] François-René Rideau. Manifeste de la libre information, 1997. `http://fare.tunes.org/Manifeste.fr.html`. 4.1

[Shivers, 1996] Olin Shivers. A Universal Scripting Framework or Lambda: the ultimate "little language". *Concurrency and Parallelism, Programming, Networking, and Security, Lecture Notes in Computer Science*, 1179:254–265, 1996. 5.2

[12] Every deserving human activity has such implications; if not, it wouldn't arouse any interest, and no one would invest one's time and other precious resources in it. Consequently, no one, by one's activities or lack thereof, as a professional, an amateur, a citizen, or a family member, can escape one's responsibilities as a human, as a living being, with respect to the resources that one engages or immobilizes, directly or indirectly.

[Turing, 1936] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 45:161–228, 1936. 3.1

[Wiener, 1957] Norbert Wiener. *Cybernetics and Society*. Houghton Mifflin, 1957. 1