

Inexcusable Complexity

Edward S. Lowry
Bedford Mass.
eslowry@alum.mit.edu
users.rcn.com/eslowry

DRAFT 17 April 2008

Abstract: Technical communication with both machines and people has been mired in inexcusable complexity for many years. The complexity results from five kinds of deficiency in language design:

- 1. Lack of structural expressiveness.**
- 2. Lack of functional expressiveness.**
- 3. Lack of language generality.**
- 4. Use of data objects which are not seriously designed to be easily arranged.**
- 5. Impracticality of expressing precise knowledge in a fully precise form.**

Progress in many areas of science, technology, and technical education may be undermined.

Complexity in technical information resulting from the following kinds of language deficiency degrade mental function, have no compensating value, and could have been substantially eliminated decades ago. Methods developed to simplify computer software can also apply to simplifying the expression of other kinds of technical knowledge.

1. Lack of structural expressiveness.

Early programming languages had little richness in their data structures. Representing the conceptual structures of richly structured applications using those data structures required use of extra data objects and that added to complexity of the expression of the application in the language. In the 1960s the "structural expressiveness" of some languages (such as C and PL/1) was improved by using a variety of "atomic" data object structures including records and pointers as well as arrays. They reduced the complexity penalty associated with mismatch between the conceptual application structures and the language data structures used to represent them. The process of adding basic data object structures to language led to excessive language complexity. It now appears possible to avoid that complexity and get better structural expressiveness by using only a single flexible data object structure. Anything less seems inexcusable.

2. Lack of functional expressiveness.

A few languages (such as Lisp, APL, SQL) had functional expressiveness. That is, they could express multiple operations on large aggregates of data in a single nested expression. They did that by using only one dominant way of stepping through the members of sets of data objects. The code for iterating through the members of a set could be implied because there was just one commonly used way to do so.

Historically, there has been a dichotomy between languages with structural and functional expressiveness. The structurally expressive languages gained their advantage by using many primitive data object structures causing many styles of iteration, which defeated functional expressiveness. That dichotomy persists until the present with currently used languages. The PROSE [1] language published by IBM in 1977 showed that both capabilities could be combined

by using flexible pointer-like data objects and just one dominant style of iteration. Complexity from decades of mismatch between programming language and data base language has been particularly harmful. By failing to combine high levels of both structural and functional expressiveness currently available languages have poor simplicity of expression compared with what was published over 30 years ago. The result has produced many disasters[2] and is increasingly inexcusable.

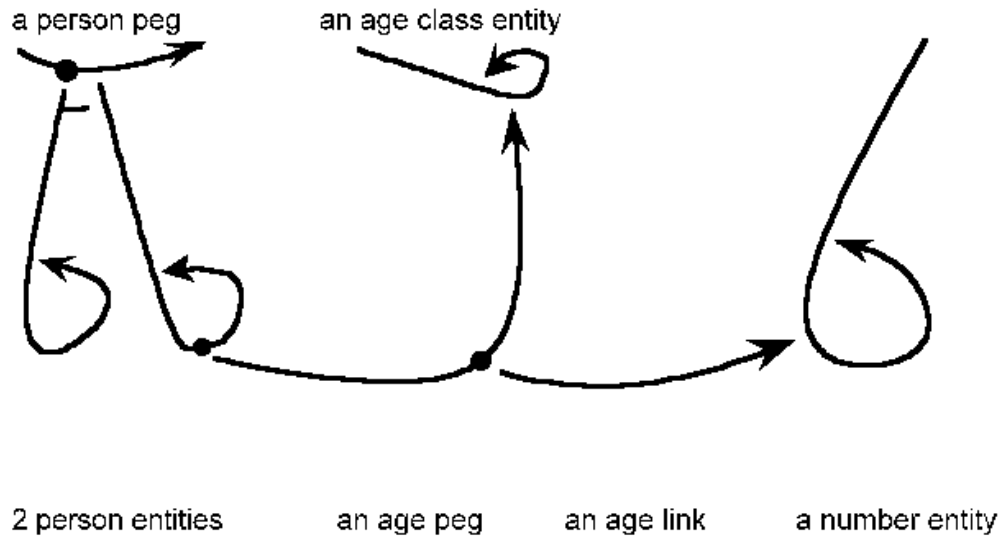
Combining the advantages clarifies future directions. It makes the potential for language generality apparent. It shows where there are and are not tradeoffs in language design. It makes further steps in simplification easier to identify. It leads to well designed basic data objects. It leads to objective comparisons between the simplicity of expressions provided by different language data models. It suggests that a newly designed general purpose language data model should be a serious candidate to be the final design.

3. Lack of language generality.

Language which combines structural and functional expressiveness can have a core semantics whose generality is largely unrestricted. The need for specialization is limited to superficial syntax. The language can use a single flexible pointer-like structure for data objects. Functions from any application area will then operate on the same kinds of data object and can easily be integrated into a single language. The lack of generality in currently used languages leads to inexcusable complexity in the need to work with multiple languages, the need to learn ephemeral languages, and the need to convert data between incompatible language data models.

4. Use of data objects which are not seriously designed to be easily arranged.

Arranging pieces of information is less complicated if they are designed to be easily arranged. They almost never are. Currently use pieces of information have traditional structures which lack any real engineering design. The value of good design is more evident in complex situations where procedures for arranging the pieces are needed and the procedures need to be kept simple. The value of good design and the designs themselves also apply to most simple situations. The following diagram shows data objects, hierarchically interconnected pointers (or needles), which have been seriously designed[3,4] to be easily arranged and rearranged.



This diagram shows how needles could be used to represent the relationship between a person and a number giving her age. Each needle points away from its parent in the hierarchy. Peg needles identify the type of objects hanging from them. The dot on the person peg connects to the first person hanging from it. The side connector on the first person gives the next person. Needles have a Tinker Toy quality which makes them easy to arrange and rearrange. A single needle can express a substantial quantity of information since it can be used to select one member from a large set.

The following sketches the rationale [4] for the design. Changes to the data structures are always discrete. The smallest changes could involve creation and deletion of objects, making and breaking connections between them, or changes to any internal state they may have. A general basic data object would then have connections to other objects and possibly internal state. The effect of eliminating various kinds of needless complexity can be described in terms of constraints imposed on the general structure of data objects. The first 3 constraints are simple. They require that the data objects be: purely connective, asymmetrical, and all have the same structure. Each of the constraints averts some kind of complexity that is arguably inexcusable.

(1). Making data objects purely connective with no internal state eliminates complexity resulting from the need to provide two language mechanisms where one is sufficient. For a truly primitive data object to have internal state, that state can do no more than play the role of specialized connections.

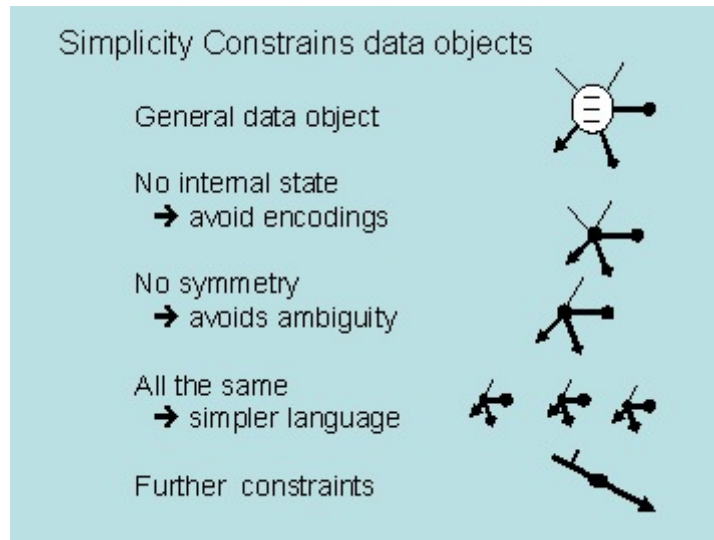
(2). Eliminating pairs of connections that are symmetrical avoids complexity which is no help since symmetrical connections would defeat the need for deterministic operation. Needs for non-deterministic operation are better provided in other ways.

(3). Using a single composite object with all the kinds of connection needed simplifies the language and does no harm since unused connections can be ignored.

(4). Combining structural and functional expressiveness requires connecting sets by a "next" connection. The possibility of part-whole relationships and sets of sets requires organizing the objects into hierarchies with connections to neighbors in the hierarchy.

(5). Empirically, hierarchy connections (including next connections), relationship connections, and type connections are all that is needed.

(6). A small simplification in working with sets of sets will result from restricting objects to have just one connection other than the hierarchy neighbor connections.



Empirically, there are very few data object structures which come close to satisfying plausible constraints imposed by simplification. Choosing among them has only small effects on the simplicity achievable or the way applications would be expressed. Contrary to conventional wisdom, the design choices have no dependency on the technical subject matter being represented as long as the subject matter is not structurally trivial. The analysis[4] completely eliminates specific kinds of harmful complexity with little need for compromise so the choices also have no dependency on how complexity is measured as long as the measure is reasonable.

In addition to being explicitly designed to be easily arranged, needles appear to be the best available design and may be a permanent optimum design, at least for information structures which are non-trivial. If so they will probably become an enduring part of the technological scenery similar to round wheels, vertical pillars, tubular pipes, flat mirrors, etc. From Airfoils to Zippers, technologists have given meticulous attention to their basic structures. For many years, information technologists have been inexcusably oblivious toward some of theirs.

5. Impracticality of expressing precise knowledge in fully precise form.

Communication with computers is necessarily complete and precise, but humans are more adaptable. For centuries technical knowledge has been presented using a mixture of precise mathematical formulas, natural language text, diagrams, etc. Much information that could be expressed precisely is not presented that way. The reader needs to distill out the precise information from the imprecise forms and integrate it. Using improved precise language, that burden can be reduced. Formal language with broad applicability can then have an enduring core set of functions. These appear to make it technically practical to develop a "universal core language supporting technical literacy". Formal language can have a durable core and flexible

open-endedness analogous to natural language. The extent to which teaching can be done effectively in that style remains to be seen[5]. The need for students to confront large amounts of precise knowledge without having it available in precise form seems increasingly inexcusable.

References

- [1] E. S. Lowry, "PROSE Specification", IBM Poughkeepsie Laboratory Technical Report TR 00.2902, Nov 1977.
- [2] E. S. Lowry, "Software Simplicity, and hence Safety - - Thwarted for Decades", 2004 International Symposium on Technology and Society. Updated at users.rcn.com/eslowry .
- [3] E. S. Lowry, "Optimum data objects for technical literacy", Educational Technology & Society, Vol 2 No 1, Jan 1999.
- [4] E. S. Lowry, "Toward Perfect Information Microstructures", users.rcn.com/eslowry Oct. 2003.
- [5] E. S. Lowry, "Formal Language as a Medium for Technical Education", Proceedings of ED-MEDIA 96, p407, AACE, June 1996. See also users.rcn.com/eslowry.
- [6] D.D. Chamberlain et al, "SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control", IBM J. Res. Development, Nov. 1976, pg 560.

Appendix: SHANNON Examples Compared with C++

To illustrate the potential for improving simplicity of expression, the following list of example expressions are provided. They indicate what degree of simplicity and clarity is achievable in a multi-purpose language and roughly what was known to be achievable in 1974 as recorded in: "PROSE Specification"[1]. It was dated 1977 but it was distributed within IBM in 1974. The language, later called Keep and Shannon, was implemented and used at Digital equipment Corporation during the 1980s and then archived. Design work is continuing.

These examples are translated from the first 7 examples given for Sequel2 (now SQL) in the IBM Journal of R&D, Nov 1976[6]. These examples showcase the benefits of including "functional expressiveness" in which iteration code is replaced and set operations for sorting, averaging, and eliminating duplicates are easily provided. For the first 7 expressions C++ uses 433 tokens (words, operators, delimiters, etc). Shannon uses 68 tokens. The Shannon expressions usually have the additional advantage of provide the result of an expression as a possible argument to an encompassing expression. The original Sequel2 code uses 84 tokens. That code is omitted as less interesting because Sequel2 is a special purpose language. Other significant comparisons would be with Java, Ada, Cobol, etc. All of the expressions are written in an informal, use-once style. Suggestions for improving the fairness of this comparison would be welcome.

Shannon

C++

Expression 1.

Names of employees in Dept. 50

name of employee of dept(50)	<pre>for (cc = 0; cc < num_employees; cc++) if (employee[cc].dept == 50) cout << employee[cc].name << endl;</pre>
------------------------------	--

Expression 2. All the different department numbers in the Employee table.

dept_no of employee condense	<pre>for (cc = 0; cc < num_depts; cc++) dept[cc].flag = 1; for (cc = 0; cc < num_employees; cc++) if (dept[employee[cc].dept].flag) { cout << employee[cc].dept << endl; dept[employee[cc].dept].flag = 0; }</pre>
------------------------------	--

Expression 3. Names of employees in Depts. 25, 47 and 53.

name of employee of every dept where 25 or 47 or 53	<pre>for (cc = 0; cc < num_employees; cc++) if (employee[cc].dept == 25 employee[cc].dept == 47 employee[cc].dept == 53) cout << employee[cc].name << endl;</pre>
---	--

Expression 4. Names of employees who work for departments in Evanston.

name of employee of dept of Evanston	<pre>for (cc = 0 ; cc < num_employees; cc++) if (department[employee[cc].dept].location == "Evanston") cout << employee[cc].name << endl;</pre>
--------------------------------------	--

Expression 5. List the employee number, name and salary of employees in Dept. 50, in order of employee number.

for employee of dept(50) minfirst empno show(empno, name, salary)	<pre>prev_num = 0; while 1 { next_num = big_number; for (cc = 0; cc < num_employees; cc++) if (employee[cc].dept == 50 && prev_num < employee[cc].number && employee[cc].number < next_num) { next_num = employee[cc].number; cc_out = cc; } if (next_num == big_number) break; cout << next_num << employee[cc_out].name << employee[cc_out].salary << endl; prev_num = next_num; }</pre>
--	---

Expression 6. Average salary of clerks.

average (salary of clerk)	<pre>for (cc = 0, sum = 0, num_clerks = 0; cc < num_employees; cc++) if (employee[cc].title == "clerk") { sum += employee[cc].salary; num_clerks++; } cout << sum / num_clerks << endl;</pre>
---------------------------	---

Expression 7. Number of different jobs held by employees in Dept.50

count job of employee of dept(50) condense	<pre>for (cc = 0; cc < num_jobs; cc++) job[cc].flag = 1; for (cc = 0, job_counter = 0; cc < num_employees; cc++) if (employee[cc].dept == 50 && job[cc].flag) { job_counter++; job[cc].flag = 0; }</pre>
--	--