Inexcusable Complexity for 40 years

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

DRAFT 24 Jun 2013

Abstract: Eliminating contamination and understanding basic structures are high priorities in many technologies. Both have been neglected in software technology and representation of technical information. Design of language for expressing precise information has been needlessly deficient for over 40 years, probably on all seven of the following leading edges:

1. **Flexibility of data structures.**
2. **Simplicity of expression.**
3. **Generality of subject matter.**
4. **Modularity of elementary information building blocks.**
5. **Fluency of precise expression.**
6. **Durability of design of core language semantics.**
7. **Breadth of potential for precise human communication.**

A false dichotomy has prevented combining wide use of simple plural expressions with flexible data structures. Incorporating iteration connections into basic data objects circumvents the dichotomy.

A significant part of software complexity is extraneous contamination resulting from language deficiencies that 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. Eliminating language-generated extraneous complexity is a process that leads to some convergences in semantics design. A needless dichotomy has caused the opposite to be widely assumed. Convergence on enduring optimum design might be approached on the above leading edges for applications which are at least moderately rich. This describes technical opportunities to improve language and the extent to which those opportunities have been neglected. Non-technical description of the neglect is given in [1,2]. Documentation of the neglect is provided in [3] where the references help show what was known and how early.

1. Flexibility of data structures.

Early programming languages had little richness in their data structures. Representing the conceptual structures of richly structured applications using those limited data structures required use of extra data objects and that added to complexity to the expression of applications. In the 1960s the flexibility of data structures of some languages (such as C and PL/1) was improved by using a variety of elementary 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 high language complexity. It is possible to avoid that complexity and get better simplicity of expression by using only a single flexible elementary data object structure.

The inclusion of integer indexed arrays as a primitive structure usually has the effect of forcing reference to integers which are irrelevant to the problem and which add complexity. Improved

flexibility was provided in the design of PROSE [4] by generalizing arrays to allow indexing by many kinds of sets, though it makes compiler optimization more challenging.

2. Simplicity of expression.

A few languages (such as APL, SQL) provided for plural expressions whose execution produces sets of objects. They could express multiple operations on large aggregates of data in a single nested expression. They did that by using a single dominant way of stepping through the members of sets of objects. The code for iterating through the members of a set could usually be implied because there was just one commonly used way to do so. Such languages gained simplicity of expression through simple plural expressions but lost simplicity of expression as a result of inflexible data structures.

Historically, there has been a dichotomy between languages which emphasized plural expressions and those which emphasized flexible data structures. The structurally rich languages gained their advantage by using many primitive data object structures. That forced the use of multiple styles of iteration, which prevented wide use of simple plural expressions. That dichotomy persists until the present with currently used languages. The PROSE [4] language effectively combined both capabilities. It used flexible pointer-like data objects to give rich structure. Extra connections on the elementary data objects (which are only implied in the documentation) provided a single dominant style of iteration.

The dichotomy has caused decades of needless mismatch between programming language and data base language that has been particularly harmful. Combining the advantages unblocks other simplifications. It allows for greater language generality. It leads to well designed basic data objects. It provides fluency that reduces dependency on supporting informal information.

Combining the advantages also clarifies language design and software development issues. It shows where there are and are not tradeoffs in language design. It makes further steps in simplification easier to identify. It leads to objective comparisons between the simplicity of expressions provided by different language data models. It suggests that a well designed general purpose language data model could have long term stability. It provides for fluent precise language that can be used in the early requirements and design phases of software development in a way that could alter our conceptualization of those phase distinctions.

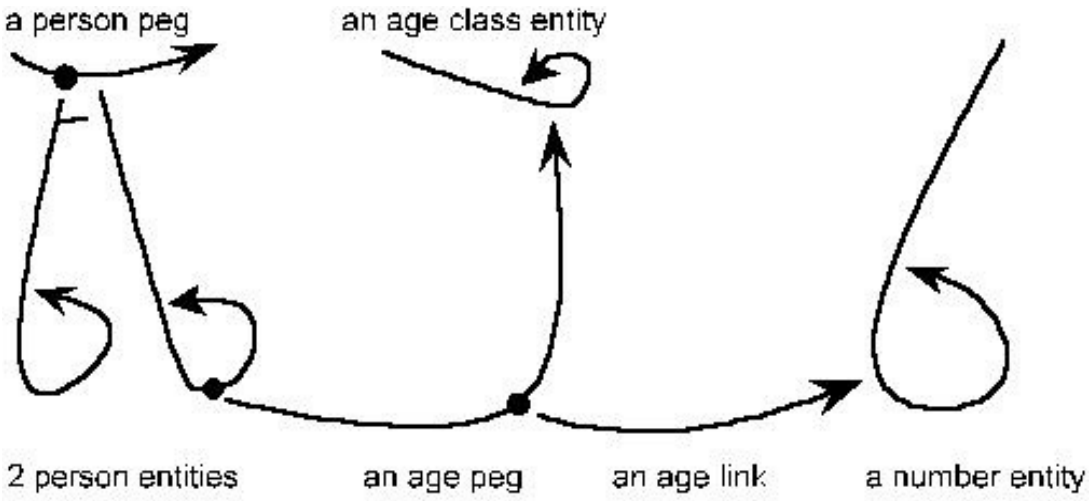
3. Generality of subject matter.

Language which combines structural richness and plural expressions can have an inner core of semantics whose precise subject matter is largely unrestricted. The need for specialization of core semantic language features becomes greatly reduced. Since all functions operate on the same flexible pointer-like data objects, functions from any application area can easily be integrated into a single language. The lack of generality in currently used languages leads to extraneous complexity, a need to learn and work with multiple languages, and a need to convert data between incompatible language data models.

4. Modularity of elementary information building blocks.

The design of elementary data objects can be said to be more “modular” than alternatives if applications using them for representation can be simpler as a result. Arranging pieces of

information is less complicated if they are well designed to be easily arranged. They almost never are. Currently used basic data objects have traditional structures which lack engineering design for high simplicity of expression. They include: text, arrays, records, lists, trees. The value of good design is compelling in complex situations where procedures for arranging the pieces need to be kept simple. However, the resulting design also applies to relatively simple situations. The following diagram shows data objects, hierarchically interconnected pointers (or needles), which have been seriously designed [5,6] to be easily arranged and rearranged.

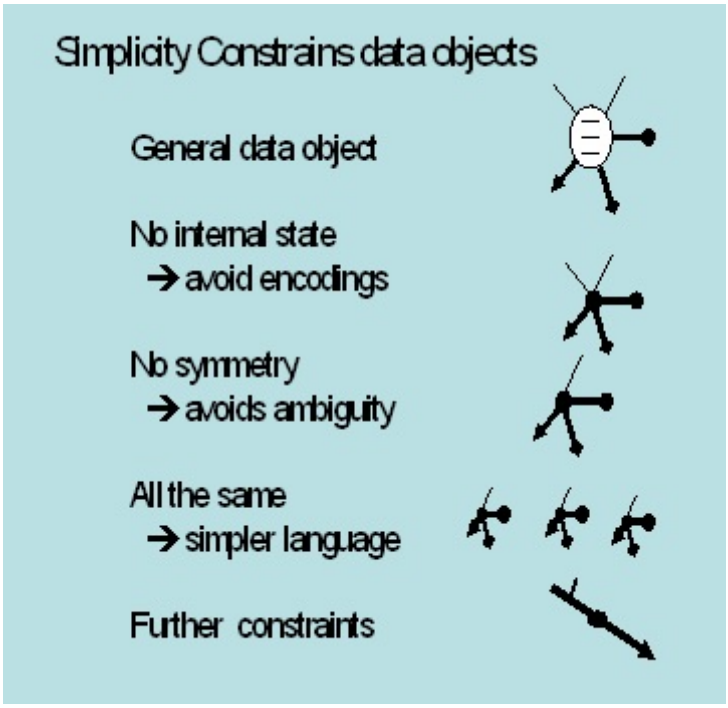


This diagram shows how needles could be used to represent the relationship between a person and a number giving her age. Each needle points away from its parent in the hierarchy. Peg needles identify the type of objects hanging from them. The dot on the person peg connects to the first person object hanging from it. The side connecter 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 [5] 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 some 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). Any symmetry in the connections would undermine deterministic execution. Needs for non-deterministic operation are better provided in other ways than providing symmetrical connections.



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

(4). Combining structural richness and plural expressions 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 and first descendant 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 some 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[5] completely eliminates specific kinds of harmful complexity with little need for compromise so the choices also have little dependency on how complexity is measured.

In addition to being explicitly designed to being easily arranged, needles appear to be the best available design and seem to approach a stable optimum design, at least for information structures which are non-trivial. If so, they may become enduring in the same way that round

wheels, vertical pillars, tubular pipes, flat mirrors, etc. are enduring. However, there is no way to mathematically characterize the set of all applications, so the optimization lacks mathematical precision. From Airfoils to Zippers, technologists have given meticulous attention to their basic structures. Information technologists and educators in technical fields have neglected some of theirs. Though not fully developed or well understood at the time, the elementary data structures in the PROSE design in 1973 seem more modular than those of currently used languages.

Students everywhere are now taught how to arrange pieces of information by educators who are unaware of the potential for pieces of information that are well-designed to be easily arranged.

5. Fluency of precise expression.

In 1982 it was possible for employees at Digital Equipment Corporation to enter an expression for computer execution such as:

6 = count every state where populatn of some city of it > 1000000

in a general programming, data base, and modeling language. Currently used wide purpose languages do not approach this level of fluency of expression, nor that of the less polished design in 1973.

Software includes complete precise instructions, and is usually accompanied by informally expressed comments, specifications and design documents. The informal material is often obscure and out of date. The need for such informal materials can be reduced by using more fluent precise language.

6. Durability of design of core language semantics.

While necessarily speculative, a case could be made that the core of precise language semantics of 2073 will bear more resemblance to the PROSE design of 1973 than that of the widely used languages of 2013. The dichotomy described above appears to have created a widespread impression that reducing language-generated extraneous complexity leads to a divergence of language designs. But that dichotomy can be eliminated. Further effort to simplify has made remaining extraneous complexity more conspicuous and isolated. Those complications raise questions about what further language complications might be justified to simplify expression but the options do not seem to cast doubt on the durability or generality of core functions. Those core functions may have long term stability while extensions and syntax may vary substantially.

7. Breadth of potential for precise human communication.

Arithmetic and algebraic notations have substantial use in human communication. Beyond that precise language for human to human communication has been very limited. Improvements discussed above in simplicity of expression, generality of subject matter, fluency, and durability of design create a prospect of more widespread use of precise language.

Traditional mathematical notations have deficiencies similar to those of computer language. They fail to combine structural richness with plural expressions and they do not use well-designed basic pieces of information. One result is that writers do not attempt to express all their ideas in

precise ways, even though much of it could be expressed precisely if more general and fluent precise language were available.

Technical knowledge is widely presented using a mixture of precise mathematical formulas, natural language text, diagrams, etc. The reader needs to distill out the precise information from the imprecise forms and mentally integrate it. Improved precise language, could reduce that burden.

Elimination of the above dichotomy reduces reasons to develop divergent design of languages. Designing highly modular elementary data objects also appears to lead to greater convergence of language design. That convergence in turn leads naturally to other convergences in design of semantic functions.

These convergences plus the improvements in simplicity, generality, fluency, and durability appear to make it technically practical to develop a language which is a reasonable candidate to become a "universal core language semantics supporting technical literacy". Formal language can have a durable core and flexible open-endedness analogous to natural language. To some extent, standards and social conventions can produce further convergences making precise information intelligible to a wide readership.

The extent to which teaching can be done effectively using such language remains to be seen[7]. There seems to be no technical justification for students to be confronted with large amounts of precise knowledge but not having it available in precise language. The language design can reflect high priorities for sound understanding of fundamental structures and avoiding contamination similar to those of other technologies.

References

[1] E. S. Lowry, “Software Simplicity, and hence Safety - - Thwarted for Decades”, 2004 International Symposium on Technology and Society.

[2] E. S. Lowry, “Technical Fluency Stifled for Decades” Jan 2010, users.rcn.com/eslowry/stifled.pdf .

[3] E. S. Lowry, “References on Software Simplification” Jun 2013, users.rcn.com/eslowry/bibliogr.pdf .

[4] E. S. Lowry, “PROSE Specification”, IBM Poughkeepsie Laboratory Technical Report TR 00.2902, Nov 1977.

[5] E. S. Lowry, "Toward Perfect Information Microstructures", users.rcn.com/eslowry/tpim.pdf Jan 2010.

[6] E. S. Lowry, "Optimum data objects for technical literacy", Educational Technology & Society, Vol 2 No 1, Jan 1999.

[7] 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/edmedium.htm .

[8] 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"[4]. It was dated 1977 but it was distributed within IBM in 1974 and a preliminary version was distributed in 1973. 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[8]. 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.

Expression 1. Names of employees in Dept. 50

Shannon	**C++**

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

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

dept_no of employee condense	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; }

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

name of employee of every dept where 25 or 47 or 53	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;

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

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

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)
	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; }

Expression 6. Average salary of clerks.

average (salary of clerk)	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;

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

count job of employee of dept(50) condense	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; }