Formal Language as a Medium for Technical Education

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

Revised September 12, 2013

Abstract

Thoroughly eliminating extraneous complexity from the expression of precise knowledge constrains the design of language for expressing it. As maximum expressive simplicity is approached, the fundamental building blocks of information are constrained toward a permanent optimum design. Using information building blocks designed that way can improve technical learning by:

- **Increasing simplicity and fluency in expression of precise knowledge.**
- **Increasing the fraction of precise knowledge expressed in precise form.**
- **Providing language for diverse precise subject matter.**
- **Providing durable design of core language semantics.**
- **Reducing the need for informal supporting information.**
- **Increasing student confidence that precise information will be decipherable.**

Students everywhere are now routinely taught how to arrange pieces of information by educators who are unaware of pieces of information that are well designed to be easily arranged. That is likely to change – disruptively. Sound teaching of computer science can guide progress toward expanded use of precise language.

Understanding basic structures and eliminating contamination and are high priorities in many technologies. Both have been neglected in language for expressing precise information. In the context of computer languages, it is possible to design building blocks of information that can be easily arranged. Little effort has been made to do so despite the fundamental need to work easily and carefully with precise information. Widely known computer languages are deficient on seven leading edges compared with a design distributed at IBM 40 years ago. See "Inexcusable Complexity for 40 years" on the web site[1]. As a result, precise information gets expressed in over-specialized ways, contaminated with extraneous complexity and poor fluency of expression. Non-technical processes contributing to the deficiencies are discussed in "Technical Fluency, Stifled for Decades" [2] on the web site.

Refinement and implementation of that design by 1982, made it possible for employees at Digital Equipment Corporation to enter an expression such as

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

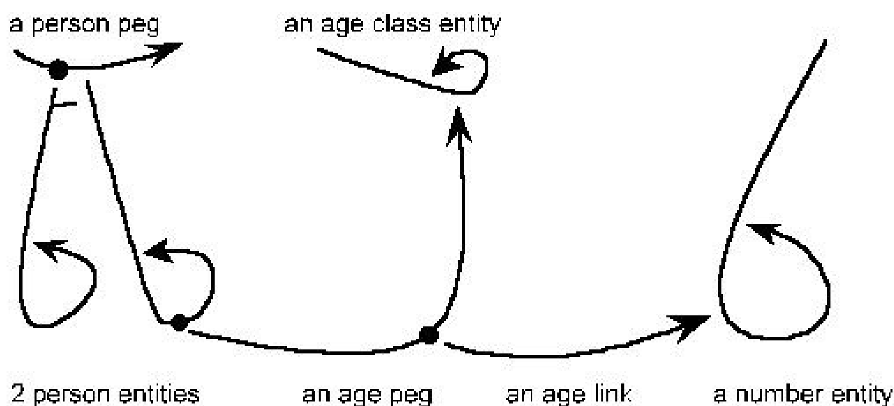
for computer execution using a general programming, data base, and modeling language. The generality, simplicity, and fluency of expression were good enough to make direct learning using such language a plausible potential across many technical subjects. However, that potential remains to be evaluated. Anyone who develops or tries to understand technical systems can benefit from literacy which includes precisely expressed ideas similar to this example.

Modularity of information building blocks

Building blocks which are easily arranged may be described as having high "modularity". Modularity can be defined so that switching to use information representations with higher modularity increases the potential for simplicity in computer programs for processing the information.

Bits and bytes have low modularity in that sense but they are preferred as basic information building blocks inside machines where they have been made fast and reliable at rapidly declining cost. Computer programs are usually written in a precise language that uses higher modularity representations which are translated into bits and bytes for machines processing.

The available evidence [3] supports a conclusion that if we do try to design building blocks with very high level of modularity, there is a convergence toward a single common design whenever the subject matter represented is structurally rich. That possibility has been little noticed because thorough simplification has been neglected. The design converges toward flexible pointer-like structures with a few additional connections for organizing the building blocks in hierarchies with lists. That design allows for expressions which freely intermix singular and plural sub-expressions in a natural language style. The same design is also adequate for simpler subject matter.



Needles representing an age relationship

As extraneous complexity is eliminated, an orderliness develops which is analogous to crystallization that develops in a liquid as heat is removed. The convergence on an enduring optimum is somewhat analogous to convergence on roundness in wheels or verticalness in pillars except that the richness of the space of applications does not lend itself to mathematical precision.

For maximum expressive simplicity all data objects will have a structure called "needles" (illustrated above) by analogy with pine needles which are pointed and organized in tree structures. Each needle points from its "parent" in the hierarchy to another object, possibly remote in the hierarchy or perhaps itself. A needle also has connections to immediate neighbors in the hierarchy, a possible next "sibling", and a possible first "child" allowing for implied iteration code and simple plural expressions.

The more important and simpler part of the analysis concludes that all data objects will have the same structure. The conclusion that such convergence happens has not been well confirmed, but even a modest attention to simplification would probably do so quickly.

While the convergence is most easily understood in the context of simplifying computer software, the conclusions apply broadly to language for simplicity in the expression of precise technical knowledge. The convergence is independent of the subject matter that is moderately rich and makes a "universal core language supporting technical literacy" technically possible.

Language generality

Since all functions operate on data structures using a common primitive object, they can be easily merged into a common language semantics regardless of what subject matter is involved. While special purpose language features are needed, they can be more easily expressed as superficial extensions to the more general purpose core language. Unspecialized language reduces barriers to accessing unfamiliar technical knowledge by reducing need for preliminary language learning.

Expanded set of stable functions

A stable set of mathematical functions often included in programming languages includes:

arithmetic including comparisons
boolean operations
set operations
matrix operations.

They have broad application and stable definitions. The improvements above provide for an expanded and integrated set of functions which can also have wide application and durability. It is possible to select groups of widely useful functions which operate on simple structures that form a language kernel.

Such groups include:

- creation and deletion of objects and relationships
- getting objects directly connected to or related to others
- arithmetic operations including comparison
- test for identicalness
- boolean operations
- conditional and case expressions
- unique selection based on key relationships (array references)
- subsetting by a selection condition

- subsetting by beginning or ending conditions
- set operations for union, intersection, difference
- testing sets for membership, inclusion, or overlap
- applying a function to each member of a set
- reduction: applying a binary function successively to the members of a set and the preceding result
- transitive closure
- sorting sets
- first order predicates over sets
- matrix operations

Almost all have been incorporated into the KEEP language developed by Digital Equipment Corporation in the 1980s. Type concepts are useful for providing diagnostics and abbreviations in the use of such functions. A simple type system would help assure language stability.

Roles for the language

Computer processing can enhance the usefulness of the language but computer assistance is not initially a requirement. Such language can serve to assist students in a variety of their basic needs:
- to access readable expositions for many mathematical, scientific, and engineering concepts.
- to communicate effectively with their teachers and others.
- to easily articulate precise descriptions of complex ideas, enhancing creativity [4,5] and problem solving ability.
- to increase productivity through easily learned technology.
- to provide a standard of information quality.

Allowing more complete expression of precise information in precise form can relieve a burden of distilling precise information from various informal representations and integrating it. Textbooks for technical subject matter express their content using precise mathematical formulas along with other kinds of less formal representation including natural language statements, metaphors, diagrams, and examples. The total explicitness provides confidence that mysteries can be resolved. Automated analysis tools can speed the resolution.

Open issues

There are many things to be learned about how improved precise language could support technical education:

- how to combine formal and informal aspects in presenting knowledge.
- how and when to introduce the language, perhaps using toy environments.
- how early students could learn directly from reading such language.
- how early students could learn to write in such language.
- how students may vary in their preference for using such language.
- how might availability of precise knowledge representation affect class discussion.

- how creativity is affected.
- how formal descriptions may be reused by students after their formal education.
- how a single language semantics may be adapted to a variety of natural language cultures.
- how multiple courses with prerequisites can be presented coherently.
- how automated analysis of prerequisites may help create customized courses quickly.
- how student need to make long term commitments to specialized learning may be affected.
- how use of rigorous language might displace traditional material that teaches rigorous thinking.

Teaching computer science, -- a way forward?

Considering the use of current computer languages when teaching computer science to young students helps focus issues. Students and others may ask:
- Why use language that was obsolescent 40 years earlier?
- Why use information building blocks which are arguably "square wheel" unreasonable?
- Are habits of elitist or obscurantist communication being imposed?
- Why expand computer science education in high schools using deficient technology.

Resolving those issues is likely to lead to use of precise language which is also suitable for expressing technical subject matter in a precise but readable form. Education leaders could develop understanding of how that process is likely to evolve and plan for it.

References

[1] E. S. Lowry, Inexcusable Complexity for 40 years, users.rcn.com/eslowry/inexcus.pdf .
[2] E. S. Lowry, Technical Fluency, Thwarted for Decades, users.rcn.com/eslowry/stifled.pdf .
[3] E. S. Lowry, Toward Perfect Information Microstructures, users.rcn.com/eslowry/tpim.pdf .
[4] E. S. Lowry, Physical Rev. pg 616, 1960, and Am. J. of Physics pg 871, 1963 For a brief description see The Electromagnetic Field in Space-time, users.rcn.com/eslowry/elmag.htm .
[5] E. S. Lowry, Proc. of ED-Media96, AACE, June 1996, pg407.(a preliminary version of this paper)

Examples

The following give descriptions of some initial content from high school chemistry, accounting and particle physics. In each case substantial amounts of precise information are presented in a precise way that was only informally expressed in the original source material. Such descriptions may be able to participate in computer executions, but not always.

Elementary CHEMISTRY

```
<<declare chemistry domain

declare element list
  has id(hydrogen, helium, lithium, ... )
  has atomic_weight in number
  has atomic_number in tally
```

```
declare atom set
  has element

declare mass quantities
declare volume quantities
declare temperature quantities
declare molecule set
  has compound
  has set atom
declare compound set
  has id(carbon_dioxide, water, molecular_oxygen, ozone, ...)
  holds set component
  has set portion converse
  has molecular_weight in number :=sum for its component take
                    its tally * atomic_weight of its element
declare component sets
  has element key
  has compound converse
  has tally
  has fraction in number
      := its tally * atomic_weight of its element / molecular_weight
                                                 of its compound
declare portion set
  mayhave compound
  has state_of_matter
  has mass
  has molecule_count in tally
  mayhave temperature
  mayhave volume

declare state_of_matter set
  has id(solid, liquid, gas)

declare transformation set
  has set input in portion
  has set output in portion
  maybe decomposition := count(its input) = 1 and count(its output) > 1
     /* gas law
certify some number satisfies every portion where gas satisfies
  its pressure * its volume / its temperate = the number
certify decomposition where compound of its input is sulphur_dioxide
  satisfies mass of its sulphur output ~= mass of its oxygen output >>
```

ACCOUNTING

This summarizes some basic accounting concepts.

```
<<declare accounting domain

declare business_entity set
        has name in string key generic
        holds set ledger in account

declare account sets
        has name in string key generic
        mayhave business_entity
        is_one_of (asset_acct, liability_acct, capital_acct)
        is_one_of (curr_asset, fixed_asset) if(asset_acct)
        is_one_of ( control_acct, subsidiary_ledger ) subtype
        holds set subsidiary_ledger if(control_acct)
        holds list acct_period

declare quarter list
        has ordinal key
        holds list journal in transaction
        has set acct_period

declare acct_period lists
        has quarter key
        has account converse
        has list entry_line := entry_line of transaction of its quarter
          where account of the acct_period = account of the entry_line
        has list debit in entry_line := its entry_line where dr
        has list credit in entry_line := its entry_line where cr
        has balance in dollar :=
                    sum (value of its debit)
                  - sum (value of its credit)
```

```
declare transaction lists
        has ordinal key
        has date
        has quarter
        has event in string
        maybe adjusting
        maybe closing
        holds set entry_line which
                (is_one_of ( dr, cr ) subtype
                has value in number
                has account)
>>
```

PARTICLE PHYSICS

```
<< declare particle_physics domain;

declare materiality set
    has id(matter, anti_matter);

declare color set
    has id(lepton, red, green, blue);
 declare tronity set
    has id(tron, trino)          /* tron implies tau, muon, or electron
    has set flavor converse;

declare generation set
    has id(first_generation, second_generation, third_generation)
    has set flavor converse;

declare flavor set
    has id( down, up, strange, charm, bottom, top)
    has generation key := first_generation if down or up else
                    second_generation if strange or charm else
                    third_generation
    has tronity key := tron if down or strange or bottom else
                                trino;

 declare handedness set
    has id(left, right);
 declare mass quantities;
```

```
declare charge values additive
   has electric in(for integer take it/3)
   mayhave weak in(for integer take it/2)
   has r_g in(for integer take it/2)
   has g_b in(for integer take it/2)
   has b_r in(for integer take it/2) := -r_g-g_b;

declare particle set
   has generation
   has tronity
   has color
   has materiality
   has mass
   has flavor := flavor(its generation, its tronity)
   has handedness
   isoneof(neutrino, tau, muon, electron, quark)
          :=    quark if not lepton
            else neutrino if trino
            else tau if bottom
            else muon if strange
            else electron
   has charge := create charge with (
               electric: (    0 if neutrino
                        else -1 if lepton
                        else 2/3 if trino
                        else -1/3 if tron )
                          *(1 if matter else -1),
               weak: 0 if left and anti_matter
                          or right and matter
                  else 1/2 if tron xor antimatter
                  else -1/2,
               r_g: (    0 if lepton or blue
                    else 1/2 if red else -1/2)
                      *(1 if matter else -1),
               g_b: (    0 if lepton or red
                    else 1/2 if green else -1/2)
                      *(1 if matter else -1)     );
>>
```