ASCILITE
SHAPING THE FUTURE OF TERTIARY EDUCATION
Show Me The Learning
Adelaide, Nov 27-30, 2016

Natural Language Proficiency and Computational Thinking: Two linked literacies of the 21st Century

Ronald Monson
Centre for Learning & Teaching
Edith Cowan University

Literacy as natural language fluency, is the primary literacy underpinning most learning but there is a new literacy gathering momentum in this information age - Computational Thinking. This paper draws connections between the two; highlighting analogs, differences, and bridges that are transforming both pedagogies while also illustrating a growing educational nexus.

Keywords: Literacies, Computational Thinking, English Language Proficiency, LACR

Introduction

What are the major literacies needed to thrive in the 21st Century? The following four are positioned as capturing key fluencies: ***L**anguage Proficiency*, ***A**rt-Design Dexterity*, ***C**omputational-Thinking Prowess*, ***R**easoning Deftness* (**LACR**). This capture also includes a further dynamic; a natural pairing of *language* and *computational thinking* working as support to another pairing, the creative literacies associated with *art-design* and *reasoning*.

The intent behind LACR's encapsulation is to promote and probe connections between existing humanities/science-like divisions in ways that recent developments suggest are becoming essential to thriving in a computer-driven society. Startling advances in machine-learning capabilities have begun to automate the acquisition of human-like *intuitions*. This is perhaps no more compellingly illustrated as in the imaginative play and learning displayed by *AlphaGo* in its recent defeat of the world's best Go players (Silver et al., 2016). Previously, automation has been about the speeding through of predictable steps but without any obvious need to invoke human-like creativity in the algorithms themselves. For example, Wing's (2014) definition of computational thinking captures this task-oriented, problem-solving nature that has characterized much of previous automation.

> Computational thinking is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry out. (J. Wing, 2014)

An updated version is called for, we suggest, in the wake of these technological bombshells; one that captures new imaginative, *intuitive* capabilities. A modern definition of computational thinking therefore, needs creativity and exploration to share top billing with problem solving and also have linguistic overtones attendant to both; further, in our view it also needs to explicitly incorporate *judgement*, *interpretation* and *collaboration*. Judgement is justified, since no computational solution, exploration or simulation succeeds without a prior rationale (including scenarios where it is *not* appropriate - ethically or feasibly). Interpretation is pivotal since, even without individual implementation, modern citizenry is increasingly required to make decisions based on the outcomes of computational thinking (consider internet search, advertising, recommendation systems and data-driven educational pathways). Next, melding global connectivity with computational thinking's signature reductionism has unleashed radical *collaborations* on grand, unprecedented scales. Finally, the recent explosion in the "Internet of Things" propounds broadening computers' laptop/desktop connotations to cover more general *computational devices*. Putting all these together, our updated version becomes:

> Computational thinking allows computational devices to solve problems, explore spaces and simulate systems judiciously, creatively and linguistically while also fostering interpretations of other's computational thinking.

Note that this is an operational definition of what computational thinking can *enable*; the thought processes it *embodies* is addressed later but first, some historical context in the evolution of computational thinking and natural language literacies.

Background

Almost 60 years ago Snow famously lamented the cultural division between The Arts and The Sciences (1993) while during the 2000's, links between coding and writing were tentatively explored without ever turning mainstream (Fernandez, 2007). There is however, in addition to the aforementioned machine-learning breakthroughs, another technological development forcing an imminent consilience - the emergence of more *literate* programming languages. While true artificial intelligence can't yet be claimed; equally, no longer can computers be considered as essentially dumb machines forever consigned to blindly following logical instructions (Nielsen, 2016). Instead, machines are beginning to display human-like capabilities for developing *intuition* (Berman, 2016) while new programming languages are enabling humans to naturally interface with such abilities. This has revolutionary implications no less in education and what may soon count as fundamental literacies.

If the development and use of acronyms reflect imperatives and priorities of an educational age, then we argue that modern digitization calls for a revised encapsulation of this era's necessary literacies. Periodic exhortations for 3R's back-to-basics or appeals to promote generic Literacy-Numeracy seem tired while more relevant groupings such as ICT (Information Computing Technology) and STEM (Science, Technology, Engineering and Mathematics) are less literacy-capture and more discipline-encapsulation as a means to promote greater inter-disciplinary integration. As laudable as this latter goal is, it has arguably come at a cost of neglecting those literacies themselves indispensable for achieving such cross-fertilization. This has perhaps been due to anointing mathematics - a vital STEM member in its own right - but in our view not the most apt choice as the unifying literacy.

As the oft-quoted *lingua franca* of science, mathematics was originally conceived as the foundational STEM literacy but not all members extensively employ its symbolism (notation and concepts from computer-science, for example, often assume more central roles). Conversely, *computational thinking* underpins almost all STEM activities as it marries science's reductionism, technology's innovation, engineering's design and mathematics' algorithms while also encompassing implementations on ubiquitous computing devices.

Another, somewhat controversial, but potentially useful recent movement has sought to expand STEM to STEAM by way of adding **A**rt/Design as a relevant and related domain (Dayton, 2014). Rather counter-productively, this enlargement often becomes mired in politics as STEM advocates resist what they see as the humanities' play for funding and influence. The issue however, is not so much whether or not creativity and design are indispensable to STEM progress and innovation - that is a given - but to what extent exposure to the arts is necessary for fostering such sensibilities. Certainly, there exists an ever-present danger of diluting STEM rigour and knowledge through over-emphasizing design but such a risk is mitigated through focusing on creative *processes* and in particular, on one central to both, *composition*.

Composing Code and Text

Striking parallels have previously been observed between composing algorithms and essays (Cummings, 2006; Fernandez, 2007) with arguably the most significant pointed out by Flower and Hayes - repeated oscillations between macro and micro viewpoints (1981). The computing macro-view corresponds to an algorithms' overall conceptualization prior to its decomposition into constituents; its micro-view implementing, testing and debugging these serving parts. A text's macro-view, on the other hand, stems from its overarching narrative, a connective thread drawing together constituent words, sentences and paragraphs into, hopefully for the author, a persuasive flow; its micro-view corresponds to the drafting and crafting of these smaller literary units in the service of this larger narrative.

Modern pedagogy points to the back and forth, the toggling, the toing and froing between these two viewpoints as most accurately characterizing the composing process for both coders and writers. The dual meaning of "compose" highlights interplays between both activities in both modes; its creative, artistic sense quintessentially evokes music or poetry but also the crafting of textual *compositions* while its (dis)aggregating sense captures the *(de)composition* so synonymous with top algorithmic design. Next, both senses are characterized in both forms.

To write effectively writers need something to say and someone to persuade. To carry out this *function* however, they need a *form* to impose structure, to give the persuasion some ballast. Part of this structure is provided by conventions specifically tuned to match the message - book genres, scholarly formatting, report layouts - but ultimately authors contribute a specific structure via headings, paragraphs, hypotheses, supporting evidence and drawn connections. This last aspect is particularly relevant given the need for good writing to consistently guide a reader's focus towards the piece's narrative while accommodating a competing tension to maintain the text's readability.

An author's *style* gives a piece its originality and is itself a complex, artistic endeavour while also being elusive to precisely categorize. Some identifiable elements of style include: an authorial voice, imagery evocation, apt noun/verb/adjective/adverb combinations, word choice, succinctness and a sentence's cadence and rhythm. A writer's development relies on adding and refining such devices however their most effective deployment comes when they dovetail with an overarching narrative.

Wing describes a "separation of concerns" (J. M. Wing, 2006, p. 33) as characterizing computational thinking while alluding to its role in distinguishing between micro and macro viewpoints, an initial practice that the most accomplished writers are able to parlay into an ensuing "joining of concerns". Designing an overall narrative commences most effectively unencumbered with stylistic concerns while conversely, bringing forth a sentence's natural rhythm can initially do without over-arching narrative impingements. Masterful expressions of both however, result from a recursive *joining* of both concerns (Flower & Hayes, 1981). A narrative benefits from readers responding to an argument pleasingly outlined, empathically-framed and compellingly articulated. On the other hand, good style benefits from an overarching narrative pointing to the "right" word or nuanced emphasis. The two also dynamically influence; the very act of stylistic improvements gives rise to deliberate changes in overall meaning and vice-versa in a virtuous cycle converging towards just *what* the author wants to say and just *how* to say it.

To code effectively, coders need a computation worth invoking. The code's *function* is the function itself while a major difference with writing is that coding's *form* comes in two flavours, the human-friendly interface used to invoke the function and the code itself. As with writing, this latter form needs a structure which is initially provided by the constructs of the chosen programming language but also by abstract *design patterns* most appropriate to the function's objective. Despite these supports, coders likewise ultimately implement their own structure in defining the sub-modules that emerge in the overarching function's *decomposition*. It is in the implementation of these modules that a coder's style begins to emerge.

A coder's style is what gives a piece of code its correctness, robustness, and readability. In contrast to writing style, aesthetic qualities give way to precision, consistency and clarity. These qualities reflect the coder's primary concern in delineating underling data structures, their unambiguous transformations all the while trying to ensure an unalloyed clarity in the program's *control flow*. This concern is so important since it allows ready *debugging* on the program's journey to correctly running.

In effect, the coder's first reader is a *compiler* who is a cold, austere entity unimpressed with adornments outside unforgiving logic. Following this initial constraint however, aesthetic demands enter the picture by way of ensuring the code's *maintainability* and *extendibility* - in short, it needs to start accommodating human readers. At this juncture, coding style assumes more literary-like connotations with questions such as - can redundancies/repetitions be removed? is there consistency and aptness in the word choice associated with function names? do functions contain humanly-graspable computational chunks? is the scope of local variables/concepts consistently displayed? are (prefix, infix and postfix) operators naturally ordered? These too speak to a coder's developing literacy and just as with writing, these local decisions about style ultimately connect with the aimed-for global functionality.

The initial value of Wing's "separation of concerns" is vital for coding as global planning is divorced from local implementations but, just as in writing, elite coders display a highly-honed facility for oscillating between holistic and immediate viewpoints. So, for example, an overall architecture can be informed by the availability of congruous sub-modules and while the process of debugging may start with localising faulty sub-functions, it often finishes with understanding the control flow as determined by the global architecture. Further, the influence is similarly bi-directional; the implementation of sub-functions frequently motivates adjustments in global architecture that can, in turn, engender remarkable simplification at the local level.

Both writing and coding exhibit similar improvement processes with *refactoring* a fundamental part of the latter. Refactoring is a technique that aims to exploit the curiously common phenomenon whereby two code-blocks can exhibit vastly different levels of readability despite implementing exactly the same algorithm. While keeping the code's functionality constant, the code's "readability" can be progressively improved to reap benefits beyond aesthetics. When done well, it can foster collaboration, programmer development, programs that run more efficiently and reliably while also helping motivate and smooth the addition of new functionality. It is also an art-form, distinguishing true artisans from hackers.

By far and away the most important technique in code refactoring is modularization whereby a chunk of code is encapsulated and replaced with a single function. Naturally that code must appear somewhere in the program to maintain functionality but it is wrapped-up, labeled, and strategically positioned elsewhere. The improvement in readability derives from now being able to conceptualize *what* the function does without concerns about *how* it does it. In so doing, a considerable cognitive overhead is removed allowing a coder to conceptualize an algorithm at the highest level.

The process of modularization is fundamental not simply as a means for organizing code but also because it forms a key part of computational thinking - the ability to conceive an algorithm, a system, a mathematical solution, almost any complex phenomena as a combination of interlocked, constituent parts. The process proffers multiple advantages which although couched here in a coding context are clearly applicable to any complex activity, as befitting a core literacy.

One of the most productive, refactoring activities is to imbue code with an almost linguistic-like readability. "Code" - the name itself indicates a space between its appearance and underlying meaning - has traditionally needed clarifying accoutrements (pseudo-code, comments, documentation) but modern languages are increasingly allowing more linguistic-like input forms (macros, operator forms, name-space management). What this means is that code-bases can be more quickly absorbed and therefore more readily maintained and extended.

The great advantage of a programming language taking on the complexion of a natural language is the resulting, enlarged space of individuality-stamped programs. Such individuality, as opposed to monolithic codebases generated by thousands, promotes coders as artisans whose ongoing improvement is motivated from learning from legendary practitioners or culturally-determined classics. Further, the additional richness of resulting programs inevitably recasts the relationship between form and function in engendering *new* algorithms.

Turning towards writing's improvement process, we initially observe that polishing or refining a piece of writing can draw upon three significant practices used in code refactoring, the first being a more precise delineation of what is being preserved between refinements. In coding what is preserved during refactoring is a program's *functionality* whereas in writing the equivalent invariant throughout the refinement is a piece's *meaning*. Already this represents is a slight divergence from coding since almost by definition re-wordings involve at least subtle shifts in meaning but nonetheless, there usually remains a faithfulness towards an overarching thread or *narrative*. Often such a narrative is said to contain a logical structure itself not unlike a program's logic defining its functionality. This structure includes reasoning chains which, if made explicit (or evaluated dynamically with real-time sentiment or coherency analysis (McNamara, Graesser, McCarthy, & Cai, 2012)), can act as guiding lodestar in satisfying the refiner that style is being improved without compromising previously established substance.

Modularization is a core component of *computational thinking* in both composing types but it is pursued relentlessly throughout coding in a way in which, if repeated with writing can yield many clarifying benefits. In this refining stage there are four types of modularization typically used as a means to shift material whose current placement may be detracting from a narrative's clarity: 1) in-text parentheses, 2) footnotes 3) appendices and 4) references. It is through liberal and systematic use of these devices that a piece of writing can be refined, filtered, *reduced* to reveal its narrative essence. Further, unifying how these are included and managed, in following the ways in which coding modules are organized (e.g. code folding), can enhance composing flexibility.

Another striking difference between the respective processes of improving writing and coding is the frequency and duration over which they take place. In writing, stand-alone compositions veer towards singly-authored, frozen-in-time artefacts. Contrast this with large codebases produced by hundreds of contributors that are often published daily following nightly *builds*. This gap suggests how writing can be made more adaptable to contemporary circumstances while also promoting writers' own development, education and perhaps even untapped virtuosity.

Literacy possesses a virtuosic hue in the sense of taking years to develop and yet being expressible within a singular "performance"; a feature that has rarely explored pedagogical implications. Consider the inefficient way students acquire essay-writing expertise: along with a final grade, a submission may receive feedback advising an improvement in word choice, the omission of redundant or repetitive terms together with establishing a more coherent and definitive narrative. Rarely however, does this end up occurring in the critiqued piece itself; instead students are left to implement (often a subset of) these recommendations in subsequent essays, where they may manifest differently in different contexts that themselves carry new literacy imperatives. An *ongoing* process however, whereby students have the opportunity to craft a piece over an extended period, would facilitate better use of feedback, incorporate new knowledge while also conveying explicit connections between literacy and localized manifestations of virtuosity.

The first rationale behind drawing connections between natural language and coding is the belief that learners with an awareness of both can ultimately become better writers and coders. Literacies are by definition lifelong processes, (in contrast to say course leaning outcomes), so ongoing opportunities present for long-term scaffolding. Further, the ongoing and rapid digitization of learning data in combination with the emerging field of learning analytics affords opportunities for verifying and shaping such longitudinal interactions.

The final rationale stems from a deeper natural language and coding nexus that is harnessing Natural Language Processing (NLP) to both *define* and *understand* algorithms in ways set to transform learning spaces. While text (completions) have revolutionized search and more recently AI-like tools such as Apple's *Siri*, Google's *Assistant* and Microsoft's *Cortana* are applying language for every-day assistance, the corresponding algorithms, while impressively summoned, all remain relatively constrained and task-oriented. Learning analytical feedback, on the other hand, is potentially on another level of complexity and importance as its algorithms define educational, life-long pathways. Consequently, the ability to understand and direct such feedback, or equivalently, understand and create algorithms in natural ways through visualization (Beheshitha, Hatala, Gašević, & Joksimović, 2016) and language (Muslim, Chatti, Mahapatra, & Schroeder, 2016) represents a new educational frontier.

Conclusion

This paper has introduced *LACR*, a grouping of four literacies aimed at reflecting a modern consilience while focussing on two, *language proficiency* and *computational-thinking prowess*. It broadened the notion of computational thinking to include recent developments in machine learning and programming languages while demonstrating how connections between the two can be used to improve both literacies. Curricula-wise, while there remains much to be done, the intent was to set the scene for perhaps an even bigger challenge, the use of these language-based means to instil LACR's other, "higher-order" literacies; *art-design dexterity* and *reasoning deftness*.

References

Beheshitha, S. S., Hatala, M., Gašević, D., & Joksimović, S. (2016). The Role of Achievement Goal Orientations When Studying Effect of Learning Analytics Visualizations. *Learning Analytics and Knowledge (LAK'16)*, 54–63. http://doi.org/10.1145/2883851.2883904

Berman, A. (2016). The Last Frontiers of AI: Can Scientists Design Creativity and Self-Awareness? Retrieved from http://singularityhub.com/2016/04/20/the-last-frontiers-of-ai-can-scientists-design-creativity-and-self-awareness/

Cummings, R. E. (2006). Coding with power: Toward a rhetoric of computer coding and composition. *Computers and Composition*, *23*(4), 430–443. http://doi.org/10.1016/j.compcom.2006.08.002

Dayton, E. (2014). *Exploring STEAM: Science, Technology, Engineering, Arts, and Mathematics*.

Fernandez, L. (2007). Code and Composition. *Ubiquity*, *8*(Issue 19), 16. http://doi.org/10.1145/1276156.1276157

Flower, L., & Hayes, J. R. J. R. (1981). A cognitive process theory of writing. *College Composition and Communication*, *32*(4), 365–387. http://doi.org/10.2307/356600

McNamara, D. S. ., Graesser, A. C. . c, McCarthy, P. M. ., & Cai, Z. . (2012). *Automated evaluation of text and discourse with Coh-Metrix*. *Automated Evaluation of Text and Discourse with Coh-Metrix*. http://doi.org/10.1017/CBO9780511894664

Muslim, A., Chatti, M. A., Mahapatra, T., & Schroeder, U. (2016). A rule-based indicator definition tool for personalized learning analytics. In *Proceedings of the Sixth International Conference on Learning Analytics & Knowledge - LAK '16* (pp. 264–273). New York, New York, USA: ACM Press. http://doi.org/10.1145/2883851.2883921

Nielsen, M. (2016). Is AlphaGo Really Such a Big Deal? | Quanta Magazine. Retrieved from https://www.quantamagazine.org/20160329-why-alphago-is-really-such-a-big-deal/

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., … Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. http://doi.org/10.1038/nature16961

Snow, C. P. (Charles P. (1993). *The two cultures*. *Nature* (Vol. 400). Cambridge University Press.

Wing, J. (2014). Computational Thinking Benefits Society. Retrieved from http://socialissues.cs.toronto.edu/index.html%3Fp=279.html

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33. http://doi.org/10.1145/1118178.1118215

Please cite as: Monson, R. (2016). Natural Language Proficiency and Computational Thinking: Two linked literacies of the 21st Century. In S. Barker, S. Dawson, A. Pardo, & C. Colvin (Eds.), *Show Me The Learning*. *Proceedings ASCILITE 2016 Adelaide* (pp. 434-439).

Note: All published papers are refereed, having undergone a double-blind peer-review process.