The Eesk Programming Language

Theron Rabe. 2013.

Natural language can concisely express complex ideas in numerous ways. It seems simple that the more ways in which a problem-solver can express potential solutions, the more likely they will find a correct solution. Even people fluent in a formal language will explain his or her ideas to another in natural language before showing someone code. This is the advantage natural language has over formal language; its users can approach a problem from many directions by altering their use of not only the language's symbols, but the rules by which those symbols carry meaning.

Symbols within a natural language, such as words in spoken-English, can combine in nearly any order to produce a syntactically valid expression (albeit with a potential of being semantically-confusing). An interpreter of a randomly-generated sequence of spoken words will, in the worst case, understand the speaker to be expressing not a high-level idea, but instead simply a list of low-level words that are, perhaps, in some way correlated. The freedom of grammar provided by spoken natural language then, seems to come from a safety mechanism that exists at both an expression (as a whole) level, and a symbol by symbol level. That is, when the interpreter realizes that the expression-level semantics of a sequence don't conform to expectations, it can safely fall back on its symbol-level rules to continue interpretation. This trait is not true of written natural language, since a properly constructed written list of words is expected to conform to certain punctuation rules. So another way of considering this is, if the interpreter were writing down everything it interpreted, it would have the ability to go back and change the punctuation it used earlier, if needed. In this sense, the all-sequence-permitting safety mechanism of natural language works by having the meaning of a high-level expression (such as a complex sentence) be either validly formed, OR simply made up of validly formed sub-expressions (which could potentially be related and re-abstracted later in time). At the low-level, all atomic symbols are safe in themselves, making the requirements of high-level validity easy to meet. Following this, it can be seen that our natural language interpreter needs to be able to “modify its punctuation” potentially at multiple and varying layers of abstraction. So, it could be said that every layer of abstraction in natural language can be referred to, represented, and modified by its interpreter.

For the sake of maximizing the number of ways in which any particular algorithm can be expressed, and therefore increasing a problem-solver's likeliness of finding correct solutions to difficult problems, and optimal (by some metric) expressions of that solution, I have looked toward recreating these expressive properties of natural language. In doing so, I have designed and implemented a new programming language Eesk, that is homoiconic at every layer of abstraction from the source text, to the abstract syntax tree, all the way down to the compiled machine code. Eesk abides as closely as possible to the safe-by-nature rules of spoken natural language that requires only validity of all sub-expressions for whole expression validity.

At the heart of Eesk is an ability to express evaluation without performing evaluation, which opens the door to writing mixtures of eagerly- and lazily- evaluated code. This, along with a powerful lexically-scoped encapsulation mechanism, lends Eesk strongly toward a purely-functional, declarative, or imperative programming paradigm. Furthermore, Eesk's machine-level homoiconicity makes it an excellent platform for writing self-modifying code, as it can dynamically generate and evaluate itself without the use of an interpreter or just-in-time compiler.