The Eesk Programming Language

Theron Rabe. 2013.

Introduction

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 be to find a correct solution. I suggest that the advantage natural language has over formal language is in its users' ability to 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.

Those symbols and rules can be broken down into simple parts. Natural and formal languages, in general, provide four fundamental elements of expression: representation, abstraction, evaluation, and encapsulation:

- **Representation** is an ability to express an atomic piece of information, such as an integer, character, or object.
- **Abstraction**, then, is the ability to form an association between a set of representations and a symbol within the language. In other words, with abstraction, the distance from atomic representation is increased. An example of abstraction, in this sense, would be when an intricate machine is given a simple name by which we associate to it: “many interconnected logic gates made from transistors” becomes simply “computer”.
- **Evaluation** is the pattern and process through which meaning is interpreted. Languages can either provide expression for evaluation explicitly, or implicitly. The sentence: “Read the following word backwards: olleh” explicitly expresses a change in evaluation that causes a reader to receive a greeting. In imperative programming languages, evaluation is usually implicitly expressed as a top-to-bottom, left-to-right calculation process that is potentially modified by control structures (such as “if” or “while”).
- **Encapsulation** is to evaluation what abstraction is to representation. An ability to abstract only helps us when we are turning our ideas into language; encapsulation offers the same benefit when we are interpreting language back into ideas. Encapsulation is the ability to group the patterns by which we determine meaning into simpler patterns.

Symbols (representations or abstractions) within a natural language, such as words in spoken-English, can combine in nearly any order to produce a syntactically valid expression. An interpreter of a randomly-generated sequence of spoken words will, in the worst case, understand the speaker to be expressing not a highly-abstracted idea, but instead simply a set of low-level representations 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 abstract semantics of a sequence don't conform to expectations, it can safely fall back on its representation-level rules to continue evaluation.

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 evaluated, 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 an abstract expression (such as a complex sentence) be either validly formed, OR made up of validly formed sub-expressions (which could potentially be related and re-abstracted later in time). I will call this safety mechanism “safe-by-nature”. This is in direct contrast to traditional safety mechanisms within formal languages, in which any expression is valid only if all its sub-expressions are valid (an “AND” to safe-by-nature's “OR”). I wall call this means of accomplishing language integrity “safe-by-rule”.

In a safe-by-nature language, all atomic symbols (representations) are safe in themselves, making the requirements of high-level validity (abstractions) easy to meet. Following this, it can be seen that our natural language interpreter needs to be able to “modify its punctuation” (change expression of evaluation) potentially at multiple and varying layers of abstraction. So, it could be said that every element of expression in a safe-by-nature language, due to its safety-biased interpreter, can interact safely without regard to the context in which they are evaluated or encapsulated.

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 safety properties of natural language. In doing so, I have designed and implemented a new programming language Eesk, that intends to conform to a safe-by-nature pattern of evaluation while abiding to as simple a grammar as possible. As an approach to this goal, Eesk intends to provide a concise and programmatic means of expressing each of the four fundamental elements of expression within a LL(1) grammar. In turn, it also attempts to be homoiconic at every layer of abstraction from the source text, to the abstract syntax tree, all the way down to the compiled machine code.

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.

Motivation

A need for Eesk originally arose from a series of experiments involving the random generation of problem-solver programs. These random programs were executed in various environments of shared resources, and those that were observed to most closely approximate a goal of longevity and homeostatic resource consumption were given the option to modify, extend, or reproduce themselves before continued competition in their environment space.

The language used to express these random problem-solvers was a simple domain-specific assembly language that, throughout the course of these experiments, was increased in complexity for the sake of enhanced expressivity. As the level of complexity and abstraction was raised, it became harder to safely generate complex programs. Over time, the problem-solvers that performed the instructions of greatest abstraction without grammatic invalidity were observed to express their implementations in patterns that approached, what I now refer to as, “safe-by-nature” expression.

Continued pursuit of highly expressive language that can be easily and safely generated for, and by, random problem-solvers has provided the foundation on which my research regarding safe-by-nature language is built. Eesk is the culmination of this work, created partly for the sake of continued investigation of these safe-by-nature patterns, and partly for the sake of exemplifying their properties.

Properties and Peculiarities of Eesk
Grammar:

- S → Representation S | Abstraction S | Decision S | Repetition S | Encapsulation S | Evaluation S | '}' S | ']' S | ')' S | {}
- Representation → number | string | operator
- Abstraction → symbol | argument | offset
- Decision → 'if' S S S
- Repetition → 'while' S S
- Encapsulation → 'Set' symbol S S | '`{' S
- Evaluation → '[' S

Symbols of Interest:

$	“Contents”	Extracts representation from an abstraction
;	“Apply”	Application of operations
:	“Store”	Abstracts a state of continuation
…	“Restore”	Restores state of continuation
=	“L-Assign”	Abstracts into a symbol, an upcoming representation
=>	“R-Assign”	Abstracts a representation into an upcoming symbol
^	“Remove”	Removes the most recent representation from the current continuation
->	“Offset”	Access from the beginning of a Set
<-	“Backset”	Access from the end of a Set
[]	“Evaluation”	Contributes a Set's members to the current continuation
`{	“Anonymous”	Anonymously encapsulates an expression
,	“Build”	Contributes the most recent representation to the dependency Set
\	“Literal”	Toggles application of safety-by-nature
#	“Comment”	Indicates a comment

Using Eesk
To begin learning Eesk, take a look at a simple example program found in manual_examples/hello.ee:

```
#This program prints some text
{
        prints “Hello, World!\n”
}
```

There's not much here to be surprised by. The # character represents a comment until the end of the line. The symbols { and } are used to delimit the program (by convention, not necessity). The third line contains an operator symbol, and an operand symbol (everything inside quotation marks is considered a single symbol). The prints operator writes text to the console, and “Hello, World!\n” provides it with that text.

To demonstrate Eesk's indifference to symbol order, take a look at manual_examples/order.ee. This program shows that the hello program described above could be written without regard to the order of operator versus operand:

```
#This program accumulates operators
{
    1 + 2 - 3;
    + 1 2 - 3;
    + - 1 2 3;
    1 2 3 + -;
}
```

When this program is compiled and executed, it will display a set of four zeros to the screen because each line remembers which operators need to be performed and holds off on using them until all required operands have been encountered, which is denoted using a ; character. Each of these lines could be more concretely written as (1 + (2 – (3))).

Special note should be taken of the fact that the output of the program is a set of four values (all zero). This indicates that values computed in Eesk are held onto in a common set (the current continuation) until they are consumed by operators. The result of any evaluation, then, is the leftovers produced by the process of computation. In imperative language, this is the equivalent of allowing functions to have multiple return values.

So far, symbols presented have all been value-typed (numbers/text). Eesk also allows user-defined symbols. Take a look at manual_examples/symbols.ee:

```
#This program has user-defined symbols
{
    result = 5 + 6 + 7 + 8;

    result$
}
```

This program calculates some additions, and represents their outcome with the symbol “result”. Since this evaluation consumes all encountered operands, we would have no output without the help of the the next line. The result symbol from earlier is re-encountered, and the value associated to that symbol is extracted using the $ operator. Were this operator not used, the line would express not the value of the additions, but the “result” symbol itself.

Notice that until “result” is actually being used, it had never been encountered. It had never been declared. Eesk assumed upon first interaction with the new symbol that the symbol's meaning was a reference to the symbol itself. From this presumption of a symbol's self-reference, along with no need to operatively consume all encountered symbols within an expression, Eesk fulfills its goal of allowing any sequence of symbols to have some valid and predictable meaning. For example, an Eesk program containing a sequence of randomly-generated symbols will (in the worst case) be interpreted as simply a Set of symbols that may (or may not) later be correlated to one another. Note that this is exactly how spoken natural language behaved under this circumstance.

As a means of relating mentioned concepts to new ideas, this next code sample will employ user-created symbols to produce a set of output values using elements borrowed from structured programming, if and while:

```
#This program uses structured programming elements
{
    while {i$ < 20} {
        if {i$ % 2} {
            i$
        } {
            prints "odd.\n"
        }
        i = i$ + 1
    }
}
```

The while structure repeats its body so long as its condition is true, as one might expect, and the if structure decides on one of two continuation routes based on the evaluation of its condition. In this case, each iteration of the loop results in either contributing the value inside the “i” symbol to the output set (if it is even), or printing some text (if it is odd). The output of this program, then, is a set of even numbers less than 20.

In the same sense that the output of an Eesk program is a set of symbols, the internal processing and evaluation of Eesk is also done in terms of sets. A “Set” is to Eesk much like what a list is to Lisp: the high-level homoiconic backbone of the language. Eesk code itself is really just a Set of instructions that are used to manipulate, construct, and deconstruct other Sets (or itself).

Eesk provides a means of programmatically definining new Sets in terms of their subsets, structures, and dependencies. Every Eesk Set has a symbol by which it is referenced, a dependency Set on whose elements its elements rely, and an instruction Set that generates or describes its belonging elements.

To better understand Sets, take a look at the example found in

TODO: keep writing documentation.

Removed these paragraphs, because they seemed out of place. Still useful, though! Save it for later...

There are four fundamental types in Eesk to correlate to the four fundamental expressive elements: values, symbols, Sets, and continuations. A value is any combination of bits that can be operated on. Examples of representable objects are numbers and strings. Symbols are Eesk's means of abstraction, and can be used to refer to any representation or abstraction (in a potentially recursive nature) in the form of a user-assigned name. Sets are the means by which encapsulation is accomplished in the language, and can be used to group data and the instructions that generate it into an evaluable container.

When a Set is evaluated, the members (or sub-expressions) associated to that Set are made available to the continuation in which the evaluation was requested. So what, then, is a continuation? Effectively, it is any expression of evaluation. In Eesk, any process of evaluation can reference and modify previous states it had encountered, much like how our natural language interpreter could modify its past use of punctuation.

For a good example to help explain the relationship between Sets and continuations, consider a program whose output relies on half of the contents of some Set. Before this Set is evaluated, the current continuation is in a state of wanting some of the Set's contents. The program can evaluate the Set (which would add its contents to the current continuation), and then recall the state of continuation from before the Set was evaluated. At this point, the program again finds itself in a state of wanting some of the Set's contents, but now, it also has those contents. Since the Set has been evaluated, and the continuation has all the data it wants (and more), it can select only the portions of that evaluation it chooses before ignoring the rest and carrying on as usual. Continuations are effectively an Eesk program's ability to go back in time and change its previous state of computation in order to achieve some goal in its current state of computation.