

# Executable English

J. Nelson Rushton, Edward Wertz  
Department of Computer Science  
Box 43104  
Lubbock TX 79409  
jrushton@ttu.edu, edward.wertz@ttu.edu

**Abstract:** *This paper introduces a declarative formal language, called Executable English, for specifying computations. The language is designed to be readable, without training, by any English speaker with an engineering or science background. Hence, programs can be read by members of an engineering team who are not themselves programmers, but whose technical knowledge is being used to create the programs. Programs in the language are fully executable, and their semantics is given by a translation to the functional language SequenceL.*

**keywords:** Controlled Languages, functional programming, executable specifications

## 1. Introduction

Our recent collaborations with Johnson Space Center's department Guidance, Navigation, and Control have revealed the need for programming languages which emphasize readability over writability. In particular, the space flight software designed at Johnson is typically designed by teams of engineers, only a few of whom are programmers. The programmers must mine the engineers for their knowledge of the systems being controlled by the proposed software, then use this knowledge to create a working program. Finally, the engineers must examine this software to verify that it meets the requirements as specified. This verification step is often a bottleneck in the process for two reasons. First, the specifications are usually implemented in procedural programming languages such as C, which are inherently difficult to verify [1]. The second problem is that the program code is, well, *code*.

This paper introduces a new programming language, Executable English (EE), which attempts to remedy this problem on both fronts. First, the language is purely declarative, in the sense of having no assignment operation. This facilitates code verifiability in general. Second, the executable "code" of the language is a fragment of "technical English" – that is, English plus symbols commonly used in non-executable technical writing. The fragment of English covered by the language is small and highly structured, so becoming an EE programmer requires a learning curve, like other languages. However, EE programs, once written, may be *read* with little or no training.

This paper is organized as follows. Section 2 describes the grammar of the language. Sections 3 and 4 describe some of its novel features – namely indefinite operators, and smart ellipses. Section 5 contains sample code written in the language. References are given in Section 6.

## 2. Grammar of terms and definitions

The grammar of terms and sentences follows below. We begin with the usual prefix notation for functions, which describes the set of names in EE.

```
literal -> number | identifier
name -> literal | prefix() | prefix(arguments) | set | vector | name_name
      | name infix name | |name|
set -> { } | {arguments}
vector -> <> | <arguments>
prefix -> sin | cos | tan | ln | log10 | floor | user-defined-prefix
infix -> + | - | * | / | ^ | mod
```

A term consists of either a name, or a quantifier followed by a noun:

```
term -> name | Quantifier noun
noun -> member of term |
Q -> some | every | no
```

Predicates and sentences are defined as follows:

```
predicate -> is adjective_schema | is not adjective_schema
           | is a noun_schema | is not a noun_schema
Sentence -> term predicate | Sentence and Sentence | Sentence or Sentence
```

Note an *adjective\_schema* or *noun\_schema* consisting of 1 or more identifiers and 0 or more variables in any order. An instance of the schema is obtained by replacing its variables with terms of appropriate type. So, for example,

```
divisor of 8 with multiplicity 3
```

is an instance of a noun, where the operator is the schema "divisor of x with multiplicity y", and the two nouns (which in this case are terms) are 8 and 3. So, an S for example could parse in the above grammar as

```
2 is a divisor of 8 with multiplicity 3.
```

Definitions of operators fit the following framework:

```
Function definiton -> term = term
noun_schema definition -> A noun is term
fact -> name is a noun.
adjective_schema definition -> variable is adjective_schema if Sentence.
```

Finally, a program is a set definitions of functions, noun schemas, facts, and adjective schemas.

### 3. Indefinite operators and backtracking

As a programming language, EE most closely fits the paradigm of a functional language, and its semantics are based on the functional language SequenceL [2]. However, it borrows two features from the logic programming paradigm which aid in giving semantics to natural-language constructs. One such feature is that of *indefinite operators*. An indefinite operator is one which operates on 0 or more arguments to return a class of values. A class is essentially a collection without order or multiplicity, but differs syntactically from a set in that it requires a quantifier to form a term.

For example, a "member of" operator is built-in operator which acts on any set  $S$  to return a class consisting of the members of  $S$ . The class can then be used with a quantifier (*every*, *some*, or *no*) to form a term. Both of the following are terms

```
every member of {1,2,3}
```

```
no member of {k | 1 ≤ k ≤ 10}
```

Note that unlike most English grammars, EE does not parse the prepositional phrase "of {1,2,3}" . Instead, "member of" is parsed as a prefix operator. which acts on the argument that follows it.

The second feature of EE borrowed from the logic programming paradigm is *backtracking*, or automated search for solutions to constraint satisfaction problems. For example. For example the code

```
A divisor of n is k satisfying 1 ≤ k ≤ n and n mod k = 0.
```

defines an indefinite operator `divisor of`. In a program containing this definition, the term "a divisor of 6" would denote the class whose members are 1, 2, 3, and 6. Note there is no explicit loop to search for candidates; this process occurs "under the hood" much as it would in Prolog.

Theoretically speaking, the condition  $1 \leq k \leq n$  is not needed in the above definition. That is, a *divisor of n* is usually defined as any  $k$  for which  $n \bmod k = 0$ , and it happens that they always fall between 1 and  $n$ . In our implementation, however, the condition  $1 \leq k \leq n$  is needed to restrict the search for candidates to a finite space. The programmer needs to know this; but the *reader* can get the correct meaning of the operator simply by knowing English, and the semantics of the well known operations  $\leq$ ,  $=$ , and  $\bmod$ . If he has no prior exposure to EE, he may wonder why the programmer included the condition  $1 \leq k \leq n$ , but it will not change the meaning for him.

### 4. Ellipses

Several languages, such as Python and Haskell, have an ellipsis construct which allows such expressions as "[1,...,5]" to evaluate to [1,2,3,4,5], or even "[2,4,...,10]" to become [2,4,6,8,10].

These constructs are clever and useful, but they fail to capture the subtly and generality with which ellipses are used in non-executable discourse. For example, a person who in ordinary discourse writes

$[2^1, \dots, 2^5]$

probably intends the five element list  $[2, 4, 8, 16, 32]$  – not the 31-element list  $[2, 3, \dots, 32]$  returned by Python and Haskell for this term. EE treats ellipses in a more flexible way. The key technical definitions involved are as follows:

If  $x$  and  $y$  are terms, an *equalizer from  $x$  to  $y$*  is an ordered pair  $(T_1, T_2)$  of terms such that  $y$  can be obtained from  $x$  by replacing all occurrences of  $T_1$  with  $T_2$ . For example,  $(\langle 5 \rangle, \langle 7 \rangle)$  is an equalizer from  $\langle 5+10 \rangle$  to  $\langle 7+10 \rangle$ . Note every pair of terms has an equalizer, since  $(x, y)$  is always an equalizer from  $x$  to  $y$ . If there is no proper subterm  $S_1$  of  $T_1$  such that  $(S_1, S_2)$  is an equalizer from  $x$  to  $y$  for some  $S_2$ , then  $(T_1, T_2)$  is said to be the *minimal equalizer from  $x$  to  $y$* . For example,  $(\langle 5 \rangle, \langle 7 \rangle)$  is the minimal equalizer from  $\langle 5+10+20 \rangle$  to  $\langle 7+10+20 \rangle$ --  $(\langle 5+10 \rangle, \langle 7+10 \rangle)$  is another equalizer in this case, but it is not minimal.

If the minimal equalizer from  $x$  to  $y$  is  $(\langle 1 \rangle, n)$  for some nonnegative integer  $n$ , then  $\langle [x, \dots, y] \rangle$  denotes the set whose  $k^{\text{th}}$  member, for all  $1 \leq k \leq n$ , is obtained from  $x$  by replacing all occurrences of  $\langle 1 \rangle$  with  $k$ . From this definition, it follows that if  $n > 1$  the  $n^{\text{th}}$  member of  $[x, \dots, y]$  will be equal to  $y$ . If  $n$  is negative then  $\langle [x, \dots, y] \rangle$  returns  $\langle [] \rangle$ . For example,

$[1, \dots, 4] = [1, 2, 3, 4]$                        $[2^1, \dots, 2^4] = [2, 4, 8, 16]$

EE can also use equalizers and ellipses with infix operations, for example,

$1 + \dots + 5 = 15$                        $2 < 4 < \dots < 100 = \text{true}$   
 $1 * 3 * \dots * 9 = 1 * 3 * 5 * 7 * 9$

## 5. Some sample code

This section presents several examples of operators defined in EE, along with explanations of the behavior of the code. The first few examples rely on the "divisor of" operator defined in Section 2.

Recall that a number is prime if it is greater than 1 and has no divisors between itself and

1. This definition in EE could be written as:

$n$  is **prime** if  $n > 1$  and no member of  $\{2, \dots, n-1\}$   
is a divisor of  $n$ .

After compiling this definition, the evaluator will, e.g., evaluate the sentence "11 is prime" to *true*, and the sentence "9 is prime" to *false*.

The code below defines the greatest common divisor (gcd) of  $m$  and  $n$  as, simply, the greatest number which is a divisor of both  $m$  and  $n$ .

```
gcd(m,n) =  
max({k | k is a divisor of m, k is a divisor of n}).
```

For example, "gcd(10,15)" would return 5, and "gcd(9,10)" would return 1. The above code would perform a brute force search for the gcd. It is well known that the Euclidean gcd algorithm is much faster, but the point here is to *specify* the function as simply as possible, in a way that is rigorous and executable. For us, performance is a secondary consideration since what our customers need is a rapidly developed, reliable *executable specification*, which also serves as a prototype. The writing of the final "production code" is seldom done in-house within NASA, but instead outsourced to contractors based on an in-house prototype.

Here is a collection of family relation operators, based on the common Prolog version:

```
A male is a member of {Reese, Jack, Ben}.
```

```
A female is a member of {Judy, Kim}.
```

```
Reese is a parent of Ben.
```

```
Kim is a parent of Ben.
```

```
Jack is a parent of Reese.
```

```
Judy is a parent of Reese.
```

```
The father of p is q satisfying
```

```
  q is a parent of p and
```

```
  q is a male.
```

```
A grandparent of p is a parent of a parent of p.
```

```
`  
A person is a male or a female.
```

```
A child of p is c satisfying
```

```
  c is a person and
```

```
  p is a parent of c.
```

```
An ancestor of p is a parent of p or a parent  
  of an ancestor of p.
```

With these definitions, the interpreter would recognize that, e.g., "the father of Ben" is *Reese*, that "some ancestor of Ben is a male" is *true*, "a child of Jack" is a class whose only member is *Reese*, etc.

## 6. References

[1] D. Cooke, M. Gelfond, N. Rushton, and H. Hu, "Application of Model-Based Technology Systems for Autonomous Systems" in *Proceedings Infotech@Aerospace*, AIAA-2005-7063 Infotech@Aerospace, Arlington, Virginia, Sep. 26-29, 2005 (8 pages).

[2] Daniel Cooke Daniel E. Cooke and J. Nelson Rushton, "SequenceL – An Overview of a Simple Language," *2005 International Conference on Programming Languages and Compilers* (PLC'05: June 27-30, 2005, Las Vegas, USA) pp. 64-70.