Latka: A Language For Random Text Generation

Getty D. Ritter
Galois, Inc.
gdritter@galois.com

Latka is a total, strongly typed functional programming language for generating random text according to predefined patterns. To this end, Latka incorporates weighted random choice as an effect in its expression language and provides a set of combinators for constructing high-level semantic constructs such as sentences and paragraphs. The eventual purpose of Latka is to produce code that can be embedded into other systems such as video games.

The primary operators of the expression language are concatenation (which is represented by juxtaposing two expressions, e.g., `e1 e2`) and choice (which is represented by a vertical bar to mimic BNF notation, e.g., `e1 | e2`), with weighted choice `m: e1 | n: e2` (where `m` and `n` are `Nats`) being a convenient syntactic sugar. Another piece of syntactic sugar is repetition, in which `n @ e` (where `n` is an expression of type `Nat`) stands for `n` repetitions of `e`. A simple program can be built out of just these primitives:

```
consonant, vowel, syllable : String
consonant = "p" | "t" | "k" | "w"
          | "h" | "m" | "n"
vowel     = "a" | "e" | "i" | "o" | "u"
syllable  = let v = 5: vowel | vowel "'" in
              5: consonant v | 2: v

-- e.g., pate'hai, aku, e'epoto'
puts (2 | 3 | 4 | 5) @ syllable
```

By default, evaluation order is call-by-name so that repeated use of the same name will result in different values, but by prefixing any binding with the keyword **`fixed`** one can specify call-by-value evaluation order, effectively selecting a value consistently for the duration of the use of the name:

```
-- Can evaluate to aa, ab, ba, or bb
puts let x = "a" | "b" in x x

-- Can only ever evaluate to aa or bb
puts let fixed x = "a" | "b" in x x
```

Latka has a set of features like other strongly typed functional languages like Haskell or OCaml, including named sums as a datatype mechanism, pattern matching, and typeclasses for ad-hoc polymorphism. Like Coq or Agda, recursive functions are restricted to structural recursion to ensure termination of embedded Latka programs. Latka also includes a set of functions for structuring larger blocks of text. These take advantage of a particular aspect of the type system which allows for variadic polymorphism. In the example below, `sent` is one such function; it takes arbitrary-length tuples of values coercible to sentence fragments and intelligently converts them to a correct sentence with capitalization and punctuation.[1]

```
samp : String * Word * Sentence
samp = ("one",wd."two",sent.("three","four"))

-- prints "One two three four."
puts sent.samp
```

Some of these examples can be seen in practice in the following program.

```
import Language.Natural.Latin as Latin

data Gender = Male | Female

pronoun : Gender -> Word
pronoun . Male   = wd."he"
pronoun . Female = wd."she"

noun : Gender -> Word
noun . Male   = wd."man"
noun . Female = wd."woman"

puts let fixed g = Male | Female in
  para.( sent.( "You see a Roman"
              , noun.g
              , "from"
              , proper_noun.(Latin/cityName)
              )
       , sent.( pronoun.g
              , "has"
              , ("brown"|"black"|"blonde")
              , "hair and carries"
              , range.50.500
              , "denarii"
              )
       )
-- It might print, for example, "You see a
-- Roman woman from Arucapa. She has black
-- hair and carries 433 denarii."
```

[1] Latka's function invocation syntax is an infixed `.` operator, borrowed from the notation Edsger W. Dijkstra outlined in *The notational conventions I adopted, and why* (2000). This operator has the highest binding power and is left-associative so that `f.x.y == (f.x).y`.