Dickinson User Guide

Vanessa McHale

Contents

Introduction

Dickinson is a text-generation language for generative literature. Each time you run your code, you get back randomly generated text.

It provides a language to define random texts like the Magical Realism Bot or fortune program.

Installing Dickinson

First, install cabal and GHC. Then:

```
cabal install language-dickinson
```

This provides `emd`, the command-line interface to the Dickinson language.

You may also wish to install manpages for reference information about `emd`. Manpages are installed at

```
emd man
```

Editor Integration

A vim plugin is available.

Program Structure

Dickinson files begin with `%-`, followed by definitions.

Example

Here is a simple Dickinson program:

```
%-

(:def main
  (:oneof
    (| "heads")
    (| "tails")))
```

Save this as `gambling.dck`. Then:

```
emd run gambling.dck
```

which will display either `heads` or `tails`.

The `:oneof` construct selects one of its branches with equal probability.

In general, when you `emd run` code, you'll see the result of evaluating `main`.

Comments

Comments are indicated with a `;` at the beginning of the line. Anything to the right of the `;` is ignored. So

```
%-
```

```
; This returns one of 'heads' or 'tails'
(:def main
  (:oneof
    (| "heads")
    (| "tails")))
```

is perfectly valid code and is functionally the same as the above.

Definitions & Names

We can define names and reference them later:

```
%-
```

```
(:def gambling
  (:oneof
    (| "heads")
    (| "tails")))
```

```
(:def main
  gambling)
```

We can `emd run` this and it will give the same results as above.

Branching

When you use `:oneof`, Dickinson picks one of the branches with equal probability. If this is not what you want, you can use `:branch`:

```
%-
```

```
(:def unfairCoin
  (:branch
    (| 1.0 "heads")
```

```
    (| 1.1 "tails")))

(:def main
  unfairCoin)
```

This will scale things so that picking `"tails"` is a little more likely.

Interpolation

We can recombine past definitions via string interpolation:

```
%-

(:def adjective
  (:oneof
    (| "beautiful")
    (| "auspicious")
    (| "cold")))

(:def main
  "What a ${adjective}, ${adjective} day!")
```

Multi-Line Strings

For large blocks of text, we can use multi-line strings.

```
(:def twain
  '''
  Truth is the most valuable thing we have - so let us economize it.
    - Mark Twain
  ''')
```

Multiline strings begin and end with `'''`.

Expressions

Branches, strings, and interpolations are expressions. A `:def` can attach an expression to a name.

```
%-

(:def color
  (:oneof
    (| "yellow")
    (| "blue")))
```

```
(:def adjective
  (:oneof
    (| "beautiful")
    (| "auspicious")
    (| color)))

(:def main
  "What a ${adjective}, ${adjective} day!")
```

Branches can contain any expression, including names that have been defined previously (such as `color` in the example above).

Lambdas

Lambdas are how we introduce functions in Dickinson.

```
(:def sayHello
  (:lambda name text
    "Hello, ${name}."))
```

Note that we have to specify the type of `name` - here, it stands in for some string, so it is of type `text`.

We can use `sayHello` with `$` (pronounced "apply").

```
(:def name
  (:oneof
    (| "Alice")
    (| "Bob")))

(:def main
  ($ sayHello name))
```

We can `emd run` this:

```
Hello, Bob.
```

`$ f x` corresponds to `f x` in ML.

Matches & Tuples

Suppose we want to randomly pick quotes. First we define a function to return a quote by Fiona Apple:

```
(:def fionaAppleQuote
  (:oneof
    (|
      '''
      "You're more likely to get cut with a dull tool than a sharp one."
```

```
      ''')
    (|
      '''
      "You forgot the difference between equanimity and passivity."
      ''')))
```

Then we can define `quote`, which returns a quote as well as the person who said it.

```
(:def quote
  (:oneof
    (| ("« Le beau est ce qu'on désire sans vouloir le manger. »", "Simone Weil"))
    (| (fionaAppleQuote, "Fiona Apple"))))
```

Each branch returns a tuple.

We can use the `:match` construct to format the result of `quote`, viz.

```
(:def formatQuote
  (:lambda q (text, text)
    (:match q
      [(quote, name)
        '''
        ${quote}
            - ${name}
        '''])))
```

```
(:def main
  $ formatQuote quote)
```

We can `emd` `run` this:

```
"You forgot the difference between equanimity and passivity."
    - Fiona Apple
```

Note the use of the `:lambda` in `formatQuote`; we specify the type (`text`, `text`).

Tags

Tags can be used to split things based on cases.

```
tydecl number = Singular | Plural
```

```
(:def indefiniteArticle
  (:lambda n number
    (:match n
      [Singular "a"]
      [Plural "some"])))
```

Note that we specify the type `number` in `(:lambda n number ...)`.

Tags themselves must begin with a capital letter while types begin with a lowercase letter.

Tags are a restricted form of sum types.

Types

REPL

To enter a REPL:

```
emd repl
```

This will show a prompt

```
emd>
```

If we have

```
%-

(:def gambling
  (:oneof
    (| "heads")
    (| "tails")))
```

in a file `gambling.dck` as above, we can load it with

```
emd> :l gambling.dck
```

We can then evaluate `gambling` if we like

```
emd> gambling
```

or manipulate names that are in scope like so:

```
emd> "The result of the coin toss is: ${gambling}"
```

We can also create new definitions:

```
emd> (:def announcer "RESULT: ${gambling}")
emd> announcer
```

Inspect the type of an expression with `:type`:

```
emd> :type announcer
text
```

We can define types in the REPL:

```
emd> tydecl case = Nominative | Oblique | Possessive
emd> :type Nominative
case
```

Saving & Restoring States

We can save the REPL state, including any definitions we've declared during the session.

```
emd> :save replSt.emdi
```

If we exit the session we can restore the save definitions with

```
emd> :r replSt.emdi
emd> announcer
```

For reference information about the Dickinson REPL:

```
:help
```

Lints

`emd` has a linter which can make suggestions based on probable mistakes. We can invoke it with `emd lint`:

```
emd lint silly.dck
```

Libraries

Dickinson allows pulling in definitions from other files with `:include`.

Using Libraries

Example

The `color` module is bundled by default:

```
(:include color)

%-

(:def main
  "Today's mood is ${color}")
```

Which gives:

```
Today's mood is citron
```

The `:include` must come before the `%-`; definitions come after the `%-`.

`color.dck` contains:

```
%-

(:def color
  (:oneof
    (| "aubergine")
    (| "cerulean")
    (| "azure")
    ...
```

Writing Libraries

Libraries can contain definitions and type declarations.

You can run `emd check` on a library file to validate it.

Scripting

`emd` ignores any lines staring with `#!`; put

```
#!/usr/bin/env emd
```

and the top of a file to use `emd` as an interpreter. As an example, here is an implementation of the Unix fortune program as a script:

```
#!/usr/bin/env emd

%-

(:def adjective
  (:oneof
    (| "good")
    (| "bad")))

(:def main
  "You will have a ${adjective} day")
```

Examples

Cowsay

Here is a variation on cowsay:

```
(:def cowsay
  (:lambda txt text
    '''

    ${txt}
    ------
          \   ^__^
           \  (oo)_____
              (__)\       )\/\
                  ||----w |
                  ||     ||
    '''))
```

Noun Declension

We can use tuples and tags to model nouns and noun declension.

```
tydecl case = Nominative | Accusative | Dative | Genitive | Instrumental

tydecl gender = Masculine | Feminine | Neuter

tydecl number = Singular | Plural

; demonstrative pronouns
; "this" or "these"
(:def decline
  (:lambda x (case, gender, number)
    (:match x
      [(Nominative, Masculine, Singular) "þes"]
      [(Accusative, Masculine, Singular) "þisne"]
      [(Genitive, (Masculine|Neuter), Singular) "þisses"]
      [(Dative, (Masculine|Neuter), Singular) "þissum"]
      [(Instrumental, (Masculine|Neuter), Singular) "þys"]
      [((Nominative|Accusative), Neuter, Singular) "þis"]
      [(Nominative, Feminine, Singular) "þeos"]
      [(Accusative, Feminine, Singular) "þas"]
      [((Genitive|Dative|Instrumental), Feminine, Singular) "þisse"]
      [((Nominative|Accusative), _, Plural) "þas"]
      [(Genitive, _, Plural) "þissa"]
```

```
      [(Dative, _, Plural) "þissum"]
    )))
```

In the REPL:

```
emd> $ decline (Nominative, Feminine, Singular)
þeos
```

This actually has no element of randomness but such capabilities are important for agreement in longer generative texts.

For guidance:

```
emd> :type decline
(-> (case, gender, number) text)
```

Shakespearen Insult Generator

Inspired by the Shakespeare Insult Kit containing a table to generate insults, we can generate our own randomly.

```
%-

(:def adjective
  (:oneof
    (| "artless")
    (| "base-court")
    (| "bawdy")
    (| "bat-fowling")
    (| "beslubbering")
    (| "beef-witted")
    (| "bootless")
    (| "beetle-headed")
    (| "churlish")
    (| "boil-brained")
    (| "cockered")
    (| "clapper-clawed")
    (| "clouted")
    (| "clay-brained")
    (| "craven")
    (| "common-kissing")
    (| "currish")
    (| "crook-pated")
    (| "dankish")
    (| "dismal-dreaming")
    (| "dissembling")
    (| "dizzy-eyed")
    (| "droning")
```

```
(| "doghearted")
(| "errant")
(| "dread-bolted")
(| "earth-vexing")
(| "elf-skinned")
(| "fawning")
(| "fobbing")
(| "froward")
(| "fat-kidneyed")
(| "frothy")
(| "fen-sucked")
(| "gleeking")
(| "flap-mouthed")
(| "goatish")
(| "fly-bitten")
(| "gorbellied")
(| "folly-fallen")
(| "impertinent")
(| "fool-born")
(| "infectious")
(| "full-gorged")
(| "jarring")
(| "guts-griping")
(| "loggerheaded")
(| "half-faced")
(| "lumpish")
(| "hasty-witted")
(| "mammering")
(| "hedge-born")
(| "mangled")
(| "hell-hated")
(| "mewling")
(| "idle-headed")
(| "paunchy")
(| "ill-breeding")
(| "pribbling")
(| "ill-nurtured")
(| "puking")
(| "knotty-pated")
(| "puny")
(| "milk-livered")
(| "qualling")
(| "motley-minded")
(| "rank")
(| "onion-eyed")
(| "reeky")
```

```
        (| "plume-plucked")
        (| "roguish")
        (| "pottle-deep")
        (| "ruttish")
        (| "pox-marked")
        (| "saucy")
        (| "reeling-ripe")
        (| "spleeny")
        (| "rough-hewn")
        (| "spongy")
        (| "rude-growing")
        (| "surly")
        (| "rump-fed")
        (| "tottering")
        (| "shard-borne")
        (| "vain")
        (| "spur-galled")
        (| "venomed")
        (| "swag-bellied")
        (| "villainous")
        (| "tardy-gaited")
        (| "warped")
        (| "tickle-brained")
        (| "wayward")
        (| "toad-spotted")
        (| "weedy")
        (| "unchin-spotted")
        (| "yeasty")
        (| "weather-bitten")))

(:def noun
  (:oneof
        (| "apple-john")
        (| "baggage")
        (| "barnacle")
        (| "bladder")
        (| "boar-pig")
        (| "bugbear")
        (| "bum-bailey")
        (| "canker-blossom")
        (| "clack-dish")
        (| "clotpole")
        (| "coxcomb")
        (| "codpiece")
        (| "death-token")
        (| "dewberry")
```

```
        (| "flap-dragon")
        (| "flax-wench")
        (| "flirt-fill")
        (| "foot-licker")
        (| "fustilarian")
        (| "giglet")
        (| "gudgeon")
        (| "haggard")
        (| "harpy")
        (| "hedge-pig")
        (| "horn-beast")
        (| "hugger-mugger")
        (| "joithead")
        (| "lewdster")
        (| "lout")
        (| "maggot-pie")
        (| "malt-worm")
        (| "mammet")
        (| "measle")
        (| "minnow")
        (| "miscreant")
        (| "moldwarp")
        (| "mumble-news")
        (| "nut-hook")
        (| "pigeon-egg")
        (| "pignut")
        (| "puttock")
        (| "pumpion")
        (| "ratsbane")
        (| "scut")
        (| "skainsmate")
        (| "strumpet")
        (| "warlot")
        (| "vassal")
        (| "whey-face")))

(:def main
  ("Thou ${adjective} ${adjective} ${noun}!"))
```

Run it get something like:

```
Thou beslubbering clouted hedge-pig!
```