

Dickinson User Guide

Vanessa McHale

Contents

Introduction	2
Installing Dickinson	2
Distributions	2
Source	2
Editor Integration	2
Tags	3
Program Structure	3
Example	3
Comments	3
Definitions & Names	4
Branching	4
Interpolation	4
Multi-Line Strings	5
Expressions	5
Lambdas	5
Matches & Tuples	6
Tags	7
Types	7
REPL	7
Saving & Restoring States	8
Builtins	8
Lints	9
Libraries	9
Using Libraries	9
Example	9
Third-Party Libraries	10
Writing Libraries	10
Scripting	10

Examples	10
Cowsay	10
Noun Declension	11
Divination Bot	12
Star Wars Name Generator	13
Shakespearean Insult Generator	14
Lyrics Bot	15
Magical Realism Bot	15

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

Distributions

Distributions for some platforms are available on the releases page.

Un-tar the package, then:

```
make install
```

Source

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.

To install with vim-plug:

```
Plug 'vmchale/dickinson' , { 'rtp' : 'vim' }
```

To automatically enable spellchecking where appropriate put

```
autocmd BufNewFile,BufRead *.dck setlocal spell spelllang=en_us
```

in your `~/.vimrc`.

Tags

To configure Dickinson with exuberant ctags or universal ctags, put the following in a file named `.ctags`:

```
--langdef=DICKINSON
--langmap=DICKINSON:.dck
--regex-DICKINSON=/:def *([[:lower:]]*[[:alnum:]]+)/\1/f,function/
--regex-DICKINSON=/tydecl *([[:lower:]]*[[:alnum:]]+)/\1/t,type/
```

I have the following in my `~/.vimrc` to keep tags updated:

```
augroup ctags
    autocmd BufWritePost *.dck :silent !ctags -R .
augroup END
```

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
```

Builtins

Dickinson has several builtin functions. You can see all names in scope (including builtins) with `:list`, viz.

```
emd> :list
oulipo
allCaps
```



```
capitalize
titleCase
```

We can inspect the type like defined names:

```
emd> :type allCaps
(-> text text)
```

Try it out:

```
emd> $ allCaps "Guilt and self-laceration are indulgences"
GUILT AND SELF-LACERATION ARE INDULGENCES
```

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")
...
```

Third-Party Libraries

Upon encountering `:include animals.mammal`, Dickinson looks for a file `animals/mammal.dck`.

When invoking `emd`, we can use the `--include` flag to add directories to search.

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 starting 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) "pes"]

[(Accusative, Masculine, Singular) "pisne"]

```
[(Genitive, (Masculine|Neuter), Singular) "pisses"]
```

[(Dative, (Masculine|Neuter), Singular) "pissum"]

```
[(Instrumental, (Masculine|Neuter), Singular) "pys"]
```

```
[(Nominative|Accusative), Neuter, Singular] "pis"]
```

[(Nominative, Feminine, Singular) "peos"]

[(Accusative, Feminine, Singular) "pas"]

```
[((Genitive|Dative|Instrumental), Feminine, Singular) "pisse"]
```

```
[(Nominative|Accusative), _, Plural) "pas"]
```

```
[(Genitive, _, Plural) "pissa"]
```

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

)))

In the REPL:

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

peos

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)
```

Divination Bot

This is a more sophisticated version of Maja Bäckvall's divination bot. The novelty is that by using tags, we get agreement between the Greek root and the definition.

```
%-
```

```
tydecl means = Fish
                | Stars
                | Snakes
                | Sun
                | Animals
                | Lips
                | Dreams
                | Placenta
                | Poo
                | Fingers
                | Number
                ...

(:def prefix
  (:lambda x means
    (:match x
      [Fish "ichthyo"]
      [Stars "astro"]
      [Snakes "ophio"]
      [Sun "helio"]
      [Animals "zoo"]
      [Lips "labio"]
      [Dreams "oneiro"]
      [Placenta "amnio"]
      [Poo "scato"]
      [Fingers "dactylo"]
      [Number "numero"]
      ...
    )))

(:def english
  (:lambda x means
    (:match x
      [Fish "fish"]
      [Stars "stars"]
      [Birds "birds"]
```

```

    [Snakes "snakes"]
    [Sun "sun"]
    [Animals "animals"]
    [Lips "lips"]
    [Dreams "dreams"]
    [Placenta "placenta"]
    [Poo "excrement"]
    [Fingers "finger movements"]
    [Number "numbers"]
    ...
  )))

(:def means
  (:pick means))

(:def postfix
  (:branch
    (| 1.0 "mancy")
    (| 0.065 "scopy")
    (| 0.03 "spication")
    (| 0.06 "logy")))

(:def main
  (:bind
    [means means]
    "{$prefix means}$postfix - divination by {$english means}"))

```

`:pick` is a builtin construct which randomly selects a tag of type `means`.

Note also `:bind` in place of `:let` — this construct resolves all randomness before bringing `means` into scope.

So the Tracery bot might produce

uranospication

Divination using the appearance of proper names.

but ours produces results like

amniomancy - divination by placenta

We've also weighted `postfix` so that the more common suffixes (such as '-mancy') occur more often.

See the full example in `examples/divinationBot.dck`

Star Wars Name Generator

As an example, consider a Star Wars name generator:

```
%-

(:def main
  (:let
    [either
      (:oneof
        (| "Quarrel")
        (| "Vult")
        ...
        (| "Blot"))]
    (:let
      [firstname
        (:oneof
          (| "Yert")
          (| "Wam")
          (| "Pommet")
          ...
          (| either))]
      [lastname
        (:oneof
          (| "Grinell")
          (| "Gorpax")
          ...
          (| either))]
      "${(:flatten firstname)} ${(:flatten lastname)}"))
```

The `:flatten` builtin makes all child outcomes equally likely. So `(:flatten firstname)` will sample “Quarrel”, “Yert” with equal probability.

See the full example in `examples/starwars.dck`

Shakespearean Insult Generator

Inspired by the Shakespeare Insult Kit’s insult table, we can generate our own insults.

```
%-

(:def adjective
  (:oneof
    (| "artless")
    (| "base-court")
    (| "bawdy")
    (| "bat-fowling")
    ...

(:def noun
```

```

(:oneof
  (| "apple-john")
  (| "baggage")
  (| "barnacle")
  (| "bladder")
  ...)

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

```

Run it get something like:

Thou beslubbering clouted hedge-pig!

See the full example in `examples/shakespeare.dck`.

Lyrics Bot

Lyrics bots sample lyrics from some particular artist; see the africa by toto bot for an example.

We can make our own Fiona Apple bot, viz.

```

%-

(:def fiona
  (:oneof
    (| "You forgot the difference between equanimity and passivity.")
    (| "You're more likely to get cut with a dull tool than a sharp one.")
    (| "The child is gone.")
    (|
      ''
      Oh darling, it's so sweet
      You think you know how crazy, how crazy I am.
      ''')
    ...))

(:def main
  fiona)

```

See the full example in `examples/fionaBot.dck`

Magical Realism Bot

We can write our own magical realism bot using builtin libraries:

```

(:include profession)
(:include geography)

```

```

%-

(:def main
  (:oneof
    (|
      (:let
        [accomplishment
          (:oneof
            (|
              (:let
                [txt
                  (:oneof
                    (| "Excel spreadsheet")
                    (| "palimpsest"))]
                [power
                  (:oneof
                    (| "comfort animals")
                    (| "practice bilocation"))]
                (:oneof
                  (| "discovers a ${txt} that allows her to ${power}"))))
            (|
              (:let
                [topic
                  (:oneof
                    (| "balneology")
                    (| "teleology")
                    (| "nephrology")
                    (| "orgonomy"))]
                "writes a monograph on ${topic}"))
            (|
              (:let
                [secret
                  (:oneof
                    (| "immortality")
                    (| "heliophagy")
                    (| "levitation")
                    (| "good skin"))]
                "discovers the secret to ${secret}"))
            ))]
        "A ${profession} in ${bigCity} ${accomplishment}"))))

```

This reuses the `bigCity` definition from the `geography` library and `profession` from the `profession` library.

This is not as sophisticated as the twitter bot but it is quite concise thanks to the libraries we used.