

# Jacinda - Functional Stream Processing Language

Vanessa McHale

## Contents

|   |          |
|---|----------|
| <b>Tutorial</b>                         | <b>2</b> |
| Tour de Force . . . . .                 | 2        |
| Patterns + Implicits, Streams . . . . . | 2        |
| Fold . . . . .                          | 3        |
| Map . . . . .                           | 3        |
| Functions . . . . .                     | 3        |
| Zips . . . . .                          | 4        |
| Scans . . . . .                         | 4        |
| Prior . . . . .                         | 4        |
| Deduplicate . . . . .                   | 5        |
| Filter . . . . .                        | 5        |
| Parting Shots . . . . .                 | 5        |
| <b>Libraries</b>                        | <b>6</b> |
| <b>Data Processing</b>                  | <b>6</b> |
| CSV Processing . . . . .                | 6        |
| Vaccine Effectiveness . . . . .         | 6        |
| <b>Machinery</b>                        | <b>7</b> |
| Functor . . . . .                       | 7        |

## Tutorial

Jacinda has fluent support for filters, maps and folds that are familiar to functional programmers; the syntax in particular is derivative of J or APL.

Jacinda is at its best when piped through other command-line tools (including `awk`).

## Tour de Force

### Patterns + Implicits, Streams

`Awk` is oriented around patterns and actions. Jacinda has support for a similar style: one defines a pattern and an expression defined by the lines that this matches, viz.

```
{% <pattern>}{<expr>}
```

This defines a stream of expressions.

One can search a file for all occurrences of a string:

```
ja '% /Bloom/}'{ '0' } -i ulysses.txt
```

`'0` here functions like `$0` in `awk`: it means the whole line.

Thus, the above functions like `ripgrep`. We could imitate `fd` with, say:

```
ls -l -R | ja '% /\.hs$/}'{ '0' }
```

This would print all Haskell source files in the current directory.

There is another form,

```
{<expr>}{<expr>}
```

where the initial expression is of boolean type, possibly involving the line context. An example:

```
{#'0>110}'{ '0' }
```

This defines a stream of lines that are more than 110 bytes (`#` is ‘tally’, it returns the length of a string).

There is also a syntax that defines a stream on *all* lines,

```
{|<expr>}
```

So `{|`0}` would define a stream of text corresponding to the lines in the file.

## Fold

Then, count lines with the word “Bloom”:

```
ja '(+)|0 {% /Bloom/}{1}' -i ulysses.txt
```

Note the *fold*, `|`. It is a ternary operator taking `(+)`, `0`, and `{% /Bloom/}{1}` as arguments. The general syntax is:

```
<expr>|<expr> <expr>
```

It takes a binary operator, a seed, and a stream and returns an expression.

## Map

Suppose we wish to count the lines in a file. We have nearly all the tools to do so:

```
(+)|0 {|1}
```

This uses aforementioned `{|<expr>}` syntax. It this defines a stream of `1`s for each line, and takes its sum.

We could also do the following:

```
(+)|0 [:1"$0
```

`$0` is the stream of all lines. `[:` is the constant operator, `a -> b -> a`, so `[:1` sends anything to `1`.

`"` maps over a stream. So the above maps `1` over every line and takes the sum.

## Functions

We could abstract away `sum` in the above example like so:

```
let val
  sum := [(+)|0 x]
in sum {% /Bloom/}{1} end
```

In Jacinda, one can define functions with a `dfn` syntax in, like in APL. We do not need to bind `x`; the variables `x` and `y` are implicit. Since `[(+)|0 x]` only mentions `x`, it is treated as a unary function.

Note also that `:=` is used for definition. The general syntax is

```
let (val <name> := <expr>)* in <expr> end
```

**Lambdas** There is syntactical support for lambdas;

```
\x. (+) | 0 x
```

would be equivalent to the above example.

## **Zips**

The syntax is:

```
, <expr> <expr> <expr>
```

One could (for instance) calculate population density:

```
, (%) $5:f $6:f
```

The postfix `:` parses the column based on inferred type; here it parses as a float.

## **Scans**

The syntax is:

```
<expr> ^ <expr> <expr>
```

Scans are like folds, except that the intermediate value is tracked at each step. One could define a stream containing line numbers for a file with:

```
(+)^0 [:1"$0
```

(this is the same as `{|ix}`)

## **Prior**

Jacinda has a binary operator, `\.`, like `q`'s `each prior` or `J`'s dyadic infix. One could write:

```
succDiff := [(-) \. x]
```

to track successive differences.

**Currying** Jacinda allows partially applied (curried) functions; one could write

```
succDiff := ((-)\.)
```

## Deduplicate

Jacinda has stream deduplication built in with the `~.` operator.

```
~.$0
```

This is far better than `sort | uniq` as it preserves order; it is equivalent to `!a[$0]++` in `awk`.

## Filter

We can filter an extant stream with `#.`, viz.

```
(>110) #. $1:i
```

`#.` takes as its left argument a unary function returning a boolean.

```
[#x>110] #. $0
```

would filter to those lines `>110` bytes wide.

## Parting Shots

```
or := [(||)|#f x]
```

```
and := [(&)|#t x]
```

```
count := [(+)|0 [:1"x]
```

`#t` and `#f` are boolean literals.

## Libraries

There is a syntax for functions:

```
fn sum(x) :=  
  (+)|0 x;  
  
fn drop(n, str) :=  
  let val l := #str  
  in substr str n l end;
```

Note the `:=` and also the semicolon at the end of the expression that is the function body.

Since Jacinda has support for higher-order functions, one could write:

```
fn any(p, xs) :=  
  (|)|#f p"xs;  
  
fn all(p, xs) :=  
  (&)|#t p"xs;
```

## Data Processing

### CSV Processing

We can process `.csv` data with the aid of `csvformat`, viz.

```
csvformat file.csv -D'|' | ja -F'\|' '$1'
```

For “well-behaved” csv data, we can simply split on `,`:

```
ja -F, '$1'
```

### Vaccine Effectiveness

As an example, NYC publishes weighted data on vaccine breakthroughs.

We can download it:

```
curl -L https://raw.githubusercontent.com/nychealth/coronavirus-data/master/latest/now-week1
```

And then process its columns with `ja`

```
ja '[1.0-x%y] {ix>1}{5:} {ix>1}{11:}' -F, -i /tmp/now-weekly-breakthrough.csv
```

As of writing:

```
0.8793436293436293
0.8524501884760366
0.8784741144414169
0.8638045891931903
0.8644207066557108
0.8572567783094098
0.8475274725274725
0.879263670817542
0.8816131830008673
0.8846732911773563
0.8974564390146205
0.9692181407757029
```

This extracts the 5th and 11th columns (discarding headers), and then computes effectiveness.

## Machinery

Under the hood, Jacinda has typeclasses, inspired by Haskell. These are used to disambiguate operators and witness with an implementation.

The language does not allow custom typeclasses.

## Functor

The map operator `"` works on all functors, not just streams. `Stream`, `List`, and `Option` are instances.