

# Jacinda - Functional Stream Processing Language

Vanessa McHale

## Contents

<b>Tutorial</b>	<b>2</b>
Language . . . . .	2
Patterns + Implicits, Streams . . . . .	2
Fold . . . . .	3
Map . . . . .	3
Functions . . . . .	4
Zips . . . . .	4
Scans . . . . .	4
Prior . . . . .	5
Deduplicate . . . . .	5
Filter . . . . .	5
Formatting Output . . . . .	5
Libraries . . . . .	6
System Interaction . . . . .	7
<b>Examples</b>	<b>7</b>
Error Span . . . . .	7
Vim Tags . . . . .	9
Enforcing Style Rules . . . . .	9
Unix Command-Line Tools . . . . .	10
grep . . . . .	10
wc . . . . .	10
head . . . . .	10

uniqu . . . . .	10
nl . . . . .	11
Data Processing . . . . .	11
CSV Processing . . . . .	11
<b>Machinery</b>	<b>12</b>
Functor . . . . .	12
IsPrintf . . . . .	12

## 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).

## Language

### 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 -1 -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.

There is also `|>`, which folds without a seed.

## 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 1s 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.

## Formatting Output

One can format output with `sprintf`, which works like `printf` in `Awk` or `C`.

As an example,

```
{|sprintf '%i: %s' (ix.'0)}
```

would display a file annotated with line numbers. Note the atypical syntax for tuples, we use `.` as a separator rather than `,`.

## 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;
```

**File Includes** One can @include files.

As an example, one could write:

```
@include'lib/string.jac'

fn path(x) :=
  intercalate '\n' (splitc x ':');

path"$0
```

intercalate is defined in lib/string.jac.

**Example** Suppose we want to mimic some functionality of sed - we'd like to replace some regular expression with a string (no capture groups, only first replacement per line)

```
@include'prelude/fn.jac'

fn replace1(re, str, line) :=
  let
    val insert := \line. \str. \ixes.
      take (ixes->1) line + str + drop (ixes->2) line
  in option line (insert line str) (match line re) end;
```

Then we could trim whitespace from a file with

```
@include'lib/sed.jac'  
  
(replace1 /\s+$/ '')"$0
```

Jacinda does not modify files in-place so one would need to use sponge perhaps:

```
ja run trimwhitespace.jac -i FILE | sponge FILE
```

## Parting Shots

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

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

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

#t and #f are boolean literals.

## System Interaction

Jacinda ignores any line beginning with #!, thus one could write a script like so:

```
#!/usr/bin/env -S ja run  
  
fn path(x) :=  
  ([x+'\n'+y])|'' (splitc x ':');  
  
path"$0
```

## Examples

### Error Span

Suppose we wish to extract span information from compiler output for editor integration. Vim ships with a similar script, `mve.awk`, to present column information in a suitable format.

```

src/Jacinda/Backend/TreeWalk.hs:319:58: error:
• The constructor 'TyArr' should have 3 arguments, but has been given 4
• In the pattern:
    TyArr _ _ (TyArr _ (TyApp _ (TyB _ TyStream) _)) _
In the pattern:
    TyArr _ _ (TyArr _ _ (TyArr _ (TyApp _ (TyB _ TyStream) _)) _)
In the pattern:
    TBuiltin (TyArr _ _
              (TyArr _ _ (TyArr _ (TyApp _ (TyB _ TyStream) _)) _))
              Fold
|
319 | eWith re i (EApp _ (EApp _ (EApp _ (TBuiltin (TyArr _ _ (TyArr _ _ (TyArr _ (TyApp _ (
|

```

To get what we want, we use `match`, which returns indices that match a regex - in our case, `/\^+/,` which spans the error location.

From the manpages, we see it has type

```
match : Str -> Regex -> Option (Int . Int)
```

```
:set fs:=\|/;
```

```
fn printSpan(str) :=
  (sprintf '%i-%i')(match str /\^+/);
```

```
printSpan:??% \|/}{'2}
```

Our program uses `|` as a file separator, thus `'2` will present us with:

```
~~~~~
```

which is exactly the relevant bit.

First, note that `"` is used to map `(sprintf '%i-%i')` over `(match ...)`. This works because `match` returns an `Option`, which is a functor. The builtin `?:` is `mapMaybe`. Thus, we define a stream

```
printSpan:??% \|/}{'2}
```

which only collects when `printSpan` returns a `Some`.



## Vim Tags

Suppose we wish to generate vim tag files for our Jacinda programs. According to `:help tags-file-format` the desired format is

```
{tagname}      {TAB} {tagfile} {TAB} {tagaddress}
```

where `{tagaddress}` is an ex command. In fact, addresses defined by regular expressions are preferable as they become outdated less quickly.

As an example, suppose we have the function declaration

```
fn sum(x) :=  
  (+) |0 x;
```

Then we need to extract `sum` and give a regex that points to where it is defined.

To do so:

```
fn mkEx(s) :=  
  '^' + s + '$/;';  
  
fn processStr(s) :=  
  let  
    val line := split s /[ \(\)+/]  
    val outLine := sprintf '%s\t%s\t%s' (line.2 . fp . mkEx s)  
  in outLine end;
```

```
processStr"%/fn +[:lower:][:latin:]*.*:=/}{'0}
```

Note the builtin `split`; according to the manpages it has type

```
split : Str -> Regex -> List Str
```

`.2` is the syntax for accessing a list - `line.2` extracts the second element.

## Enforcing Style Rules

Suppose our style guide says that lines can be at most 80 characters. We can show any such lines we've introduced with:

```
git diff origin/master | ja '[#x>81]#.{%/^\\+}{'}
```

(81 to allow for the leading +)

## Unix Command-Line Tools

To get a flavor of Jacinda, see how it can be used in place of familiar tools:

### grep

```
ja '%/the/}'{0}' -i FILE
```

### wc

To count lines:

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

or

```
[y]|0 {|ix}
```

To count bytes in a file:

```
(+)|0 [#x+1]"$0
```

or

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

### head

To emulate `head -n60`, for instance:

```
{ix<=60}'{0}
```

### uniq

```
fn step(acc, this) :=  
  if this = acc->1  
    then (this . None)  
    else (this . Some this);
```

```
(->2):?step^(''.None) $0
```

This tracks the previous line in a state and only adds the current line to the stream if it is different.

**nl**

We can emulate `nl -b a` with:

```
{|sprintf ' %i %s' (ix.'0)}
```

To count only non-blank lines:

```
fn empty(str) :=
  #str = 0;

fn step(acc, line) :=
  if empty line
  then (acc->1 . '')
  else (acc->1 + 1 . line);

fn process(x) :=
  if !empty (x->2)
  then sprintf ' %i\t%s' x
  else '';

process"step^(0 . '')" $0
```

## 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-week
```

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

## IsPrintf

The `IsPrintf` typeclass is used to type `sprintf`; strings, integers, floats, booleans, and tuples of such are members.

```
sprintf '%i' 3
```

and

```
sprintf '%s-%i' ('str' . 2)
```

are both valid.