

Jacinda - Functional Stream Processing Language

Vanessa McHale

Contents

Tutorial	2
Language	2
Patterns + Implicits, Streams	2
Fold	3
Custom Field Separators	3
Map	3
Functions	3
Zips	4
Scans	4
Prior	4
Deduplicate	5
Filter	5
Formatting Output	5
Reporting	5
Libraries	5
System Interaction	6
Define Values on the Command-Line	7
Examples	8
File Sizes	8
Vim Tags	8
Error Span	9
Unix Command-Line Tools	10
wc	10
head	10
basename	10
tr	10
uniq	11
nl	11
Extract Source from Cabal	11
Make Recipe: Format	12
Fixity Declarations for HLint	12
Data Processing	12

CSV Processing	12
Machinery	13
Typeclasses	13
Functor	13
IsPrintf	13
Row Types	13

Tutorial

Jacinda is well-suited to processing the output of Unix tools: regular expressions scan for relevant output and one can split on separators.

There is additionally support for filters, maps and folds that are familiar to functional programmers.

Language

Patterns + Implicits, Streams

In Jacinda, one writes a pattern and an expression defined on matching lines, 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. So this would print all lines that match the pattern `Bloom`.

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

To 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.

Custom Field Separators

Like AWK, Jacinda allows us to define custom field separators:

```
printenv | ja -F= '{% /^PATH/}{`2}'
```

This splits on `=` and matches lines beginning with `PATH`, returning the second field—in this case, the value of `PATH`.

Map

Suppose we wish to count the lines in a file.

```
(+)|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, $a \rightarrow b \rightarrow a$, 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.

`[y]` is treated as binary. Thus, `[y]▷$0` prints the last line.

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 `[(+)|0 x]`.

Zips

The syntax is:

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

One could (for instance) calculate population density:

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

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 `,`.

Reporting

One can print a stream and a summary value (usually the result of a fold):

```
$1 $> (+)|0 $1:
```

Try:

```
seq 10000 | ja '$1 $> (+)|0 $1:'
```

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.
```

In-Place File Modification We could trim whitespace from lines with:

```
(sub1 /\s+$/ 0)"$0

sub1 is like AWK's sub and only substitutes the first occurrence. 0 is zilde, and
can be used to represent an empty string or vector.

Jacinda does not modify files in-place so one would need to use sponge, viz.

ja '(sub1 /\s+$/ 0)"$0' -i FILE | sponge FILE
```

Prelude

```
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
```

Define Values on the Command-Line

We can jerry-rig a RIS to .bib converter:

```
:set fs:=/ -\s*/;

fn bib(ty) :=
  ?ty='JOUR';'article'
  ;?ty='BOOK';'book'
  ;?ty='CONF';'inproceedings'
  ;'misc';

fn field(r) :=
  ?r='AU';Some 'author'
  ;?r='PY';Some 'year'
  ;?r='TI';Some 'title'
  ;?r='VL';Some 'volume'
  ;?r='JO';Some 'journal'
  ;?r='DO';Some 'doi'
  ;None;

.?{ | ?`1='TY';Some ('@'+bib `2+'{'+'name+',')
  ;?`1='ER';Some '}'
  ;?`1='UR';Some (' url={\\url{'+'2+'}},')
  ;[' '+x+'={'+'2+'},''](field `1)}
```

Running this on its own will fail:

```
ja: 22:36 'name' is not in scope.
```

We can specify name per-invocation like so:

```
> ja run ris2bib.jac -i shannon.ris -Dname='shannon1948'
@article{shannon1948,
  author={Shannon, Claude E.},
  year={1948},
  title={A Mathematical Theory of Communication},
  volume={27},
}
```

Examples

File Sizes

To find the total size of files in a directory:

```
ls -l | ja '(+)|0 {ix>1}{`5:}'  
79769
```

We can define `prettyMem` as a library function, viz.

```
fn prettyMem(x) :=  
  ?x ≥ 1073741824.0  
  ;sprintf'%f.2 GB' (x%1073741824.0)  
  ;?x ≥ 1048576.0  
  ;sprintf'%f.2 MB' (x%1048576.0)  
  ;?x ≥ 1024.0  
  ;sprintf'%f.2 kB' (x%1024.0)  
  ;sprintf'%f.0 b' x;
```

The `%f.2` format specifier limits output to two digits after the decimal point.

Then:

```
ls -l | ja "@include'lib/prefixSizes.jac' prettyMem((+)|0.0 {ix>1}{`5:})"  
77.89 kB
```

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)
```



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

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

Error Span

```
src/Jacinda/Backend/TreeWalk.hs:319:58: error:
```

- ```
TyArr _ _ (TyArr _ (TyApp _ (TyB _ TyStream) _)) _
```

```
TyArr _ _ (TyArr _ _ (TyArr _ (TyApp _ (TyB _ TyStream) _)) _)
```

```
TBuiltin (TyArr _ _
 (TyArr _ _ (TyArr _ (TyApp _ (TyB _ TyStream) _)) _))
Fold
```

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

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

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`.

## Unix Command-Line Tools

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

### **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}
```

### **basename**

```
fn fileName(x) :=
 x ~* 2 /([^\/*\])(.*)/;
```

will remove the directory part of a filename.

### **tr**

We can present the `PATH` with

```
echo $PATH | tr ':' '\n'
```

To do so in Jacinda, we use `:` as field separator, viz.

```
echo $PATH | ja -F: "{[x+'\n'+y]|>\`$}"
```

``$` is all fields in a line, as a list.

## 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
```

We could write `process` as

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

using the laconic syntax for conditionals, `?<bool>;<expr>;<expr>`

## Extract Source from Cabal

We can use

```
ja -F'\s*:\s*' '{%/hs-source-dirs/}{`2}' -i jacinda.cabal
```

to extract all source directories from a `.cabal` file: executables, test suites, &c.

This can be combined with `fd` to search for all Haskell source files defined by a `.cabal` file, viz.

```
fd '\.(cpphs|hs)$' $(ja -F'\s*:\s*' '{%/hs-source-dirs/}{`2}' -i jacinda.cabal)
```

## Make Recipe: Format

We can define a make recipe `fmt` to format all Haskell files:

```
fmt:
```

```
fd '\.(cpphs|hs)$' $(ja -F'\s*:\s*' '{%/hs-source-dirs/}{`2}' -i apple.cabal) -x stylish-haskell
```

## Fixity Declarations for HLint

To extract fixity declarations and present them in a format suitable for HLint:

```
ja "{%/infix(r|l)? \d+/{sprintf '- fixity: %s' \0}" -i src/FILE.hs
```

We can define a recipe `fix` to extract all fixity definitions:

```
fix:
```

```
fd '\.(cpphs|hs|x|y|hsc)$' $(ja -F'\s*:\s*' '{%/hs-source-dirs/}{`2}' -i apple.cabal) -x ja
```

Note that this works on Happy, Alex, etc. source files.

## Data Processing

### CSV Processing

**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-weekly-breakthrough.csv -o /tmp/now-weekly-breakthrough.csv
```

And then process its columns with `ja` using CSV mode:

```
ja --csv '[1.0-x%y] {ix>1}{`5:} {ix>1}{`11:}' -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.

**Inflation** We start with New Zealand's food price index:

```
curl -O https://www.stats.govt.nz/assets/Uploads/Food-price-index/Food-price-index-September-2023/Download-data/food-price-index-september-2023-weighted-average-prices.csv
```

Then:

```
ja --csv '(%)\. {%/Apple/}{`3:}'
```

## Machinery

### Typeclasses

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.

### 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.

### Row Types

The  $\rightarrow n$  accessors work on all applicable tuples, so

```
(a.b.c)->2
```

and

```
(a.b)->2
```

are both valid.

Moreover,

$(a.b) \rightarrow 3$

will be caught during typechecking.