Jacinda - Functional Stream Processing Language

Vanessa McHale

Contents

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

Filtering

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.

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 {#t}{1}
```

This uses aforementioned `{<expr>}{<expr>}` syntax. `#t` is a boolean literal. So 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.

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 defines functions like dfns 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 function definition. The general syntax is

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

Zips

The syntax is:

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

One could (for instance) calculate population density:

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

The postfix `:f` parses the column as an integer.

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 `{#t}{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.

Parting Shots

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

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

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

Machinery

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