jq manual (excerpt)

A jq program is a “filter”: it takes an input, and produces an output. There are a lot of builtin filters for extracting a particular field of an object, or converting a number to a string, or various other standard tasks.

Filters can be combined in various ways - you can pipe the output of one filter into another filter, or collect the output of a filter into an array.

Some filters produce multiple results, for instance there’s one that produces all the elements of its input array. Piping that filter into a second runs the second filter for each element of the array. Generally, things that would be done with loops and iteration in other languages are just done by gluing filters together in jq.

It’s important to remember that every filter has an input and an output. Even literals like “hello” or 42 are filters - they take an input but always produce the same literal as output. Operations that combine two filters, like addition, generally feed the same input to both and combine the results. So, you can implement an averaging filter as `add / length` - feeding the input array both to the `add` filter and the `length` filter and then performing the division.

But that’s getting ahead of ourselves. :) Let’s start with something simpler:

Types and Values

jq supports the same set of datatypes as JSON - numbers, strings, booleans, arrays, objects (which in JSON-speak are hashes with only string keys), and “null”.

Booleans, null, strings and numbers are written the same way as in JSON. Just like everything else in jq, these simple values take an input and produce an output - `42` is a valid jq expression that takes an input, ignores it, and returns 42 instead.

Numbers in jq are internally represented by their IEEE754 double precision approximation. Any arithmetic operation with numbers, whether they are literals or results of previous filters, will produce a double precision floating point result.

However, when parsing a literal jq will store the original literal string. If no mutation is applied to this value then it will make to the output in its original form, even if conversion to double would result in a loss.

Array construction: `[]`

As in JSON, `[]` is used to construct arrays, as in `[1,2,3]`. The elements of the arrays can be any jq expression, including a pipeline. All of the results produced by all of the expressions are collected into one big array. You can use it to construct an array out of a known quantity of values (as in `[.foo, .bar, .baz]`) or to "collect" all the results of a filter into an array (as in `[.items[].name]`)

Once you understand the "," operator, you can look at jq's array syntax in a different light: the expression `[1,2,3]` is not using a built-in syntax for comma-separated arrays, but is instead applying the `[]` operator (collect results) to the expression 1,2,3 (which produces three different results).

If you have a filter `X` that produces four results, then the expression `[X]` will produce a single result, an array of four elements.

Examples

Filter	`[.user, .projects[]]`
Input	`{"user":"stedolan", "projects": ["jq", "wikiflow"]}`
Output	`["stedolan", "jq", "wikiflow"]`
Run	

Filter	`[ .[] | . * 2]`
Input	`[1, 2, 3]`
Output	`[2, 4, 6]`
Run	

Object Construction: `{}`

Like JSON, `{}` is for constructing objects (aka dictionaries or hashes), as in: `{"a": 42, "b": 17}`.

If the keys are "identifier-like", then the quotes can be left off, as in `{a:42, b:17}`. Variable references as key expressions use the value of the variable as the key. Key expressions other than constant literals, identifiers, or variable references, need to be parenthesized, e.g., `{("a"+"b"):59}`.

The value can be any expression (although you may need to wrap it in parentheses if, for example, it contains colons), which gets applied to the {} expression's input (remember, all filters have an input and an output).

```
{foo: .bar}
```

will produce the JSON object `{"foo": 42}` if given the JSON object `{"bar":42, "baz":43}` as its input. You can use this to select particular fields of an object: if the input is an object with “user”, “title”, “id”, and “content” fields and you just want “user” and “title”, you can write

```
{user: .user, title: .title}
```

Because that is so common, there’s a shortcut syntax for it: `{user, title}`.

If one of the expressions produces multiple results, multiple dictionaries will be produced. If the input’s

```
{"user":"stedolan","titles":["JQ Primer", "More JQ"]}
```

then the expression

```
{user, title: .titles[]}
```

will produce two outputs:

```
{"user":"stedolan", "title": "JQ Primer"}
{"user":"stedolan", "title": "More JQ"}
```

Putting parentheses around the key means it will be evaluated as an expression. With the same input as above,

```
{(.user): .titles}
```

produces

```
{"stedolan": ["JQ Primer", "More JQ"]}
```

Variable references as keys use the value of the variable as the key. Without a value then the variable’s name becomes the key and its value becomes the value,

```
"f o o" as $foo | "b a r" as $bar | {$foo, $bar:$foo}
```

produces

```
{"foo":"f o o","b a r":"f o o"}
```

Examples

Filter	`{user, title: .titles[]}`
Input	`{"user":"stedolan","titles":["JQ Primer", "More JQ"]}`
Output	`{"user":"stedolan", "title": "JQ Primer"}`
	`{"user":"stedolan", "title": "More JQ"}`

Run	

Filter	`{(.user): .titles}`
Input	`{"user":"stedolan","titles":["JQ Primer", "More JQ"]}`
Output	`{"stedolan": ["JQ Primer", "More JQ"]}`
Run	

Recursive Descent: `..`

Recursively descends `.`, producing every value. This is the same as the zero-argument `recurse` builtin (see below). This is intended to resemble the XPath `//` operator. Note that `..a` does not work; use `.. | .a` instead. In the example below we use `.. | .a?` to find all the values of object keys “a” in any object found “below” `..`.

This is particularly useful in conjunction with `path(EXP)` (also see below) and the `?` operator.

Examples

Filter	`.. | .a?`
Input	`[[{"a":1}]]`
Output	`1`
Run	

Conditionals and Comparisons

==, !=

The expression ‘a == b’ will produce ‘true’ if the results of evaluating a and b are equal (that is, if they represent equivalent JSON values) and ‘false’ otherwise. In particular, strings are never considered equal to numbers. In checking for the equality of JSON objects, the ordering of keys is irrelevant. If you’re coming from JavaScript, please note that jq’s == is like JavaScript’s ===, the “strict equality” operator.

!= is “not equal”, and ‘a != b’ returns the opposite value of ‘a == b’

Examples

Filter	`. == false`
Input	`null`
Output	`false`
Run	

Filter	`. == {"b": {"d": (4 + 1e-20), "c": 3}, "a":1}`
Input	`{"a":1, "b": {"c": 3, "d": 4}}`
Output	`true`
Run	

Filter	`.[] == 1`
Input	`[1, 1.0, "1", "banana"]`
Output	`true`
	`true`
	`false`
	`false`
Run	

if-then-else-end

`if A then B else C end` will act the same as `B` if `A` produces a value other than false or null, but act the same as `C` otherwise.

`if A then B end` is the same as `if A then B else . end`. That is, the `else` branch is optional, and if absent is the same as `.`. This also applies to `elif` with absent ending `else` branch.

Checking for false or null is a simpler notion of "truthiness" than is found in JavaScript or Python, but it means that you'll sometimes have to be more explicit about the condition you want. You can't test whether, e.g. a string is empty using `if .name then A else B end`; you'll need something like `if .name == "" then A else B end` instead.

If the condition `A` produces multiple results, then `B` is evaluated once for each result that is not false or null, and `C` is evaluated once for each false or null.

More cases can be added to an if using `elif A then B` syntax.

Examples

Filter	`if . == 0 then "zero" elif . == 1 then "one" else "many" end`
Input	`2`
Output	`"many"`
Run	

>, >=, <=, <

The comparison operators `>`, `>=`, `<=`, `<` return whether their left argument is greater than, greater than or equal to, less than or equal to or less than their right argument (respectively).

The ordering is the same as that described for `sort`, above.

Examples

Filter	`. < 5`
Input	`2`
Output	`true`
Run	

`and, or, not`

jq supports the normal Boolean operators `and`, `or`, `not`. They have the same standard of truth as if expressions - `false` and `null` are considered “false values”, and anything else is a “true value”.

If an operand of one of these operators produces multiple results, the operator itself will produce a result for each input.

`not` is in fact a builtin function rather than an operator, so it is called as a filter to which things can be piped rather than with special syntax, as in `.foo and .bar | not`.

These three only produce the values `true` and `false`, and so are only useful for genuine Boolean operations, rather than the common Perl/Python/Ruby idiom of “value_that_may_be_null or default”. If you want to use this form of “or”, picking between two values rather than evaluating a condition, see the `//` operator below.

Examples

Filter	`42 and "a string"`
Input	`null`
Output	`true`
Run	

Filter	`(true, false) or false`
Input	`null`
Output	`true`
	`false`
Run	

Filter	`(true, true) and (true, false)`
Input	`null`
Output	`true`

	`false`
	`true`
	`false`
Run	

Filter	`[true, false | not]`
Input	`null`
Output	`[false, true]`
Run	

Alternative operator: `//`

The `//` operator produces all the values of its left-hand side that are neither `false` nor `null`, or, if the left-hand side produces no values other than `false` or `null`, then `//` produces all the values of its right-hand side.

A filter of the form `a // b` produces all the results of `a` that are not `false` or `null`. If `a` produces no results, or no results other than `false` or `null`, then `a // b` produces the results of `b`.

This is useful for providing defaults: `.foo // 1` will evaluate to `1` if there's no `.foo` element in the input. It's similar to how `or` is sometimes used in Python (jq's `or` operator is reserved for strictly Boolean operations).

`some_generator // defaults_here` is not the same as `some_generator | . // defaults_here`. The latter will produce default values for all non-`false`, non-`null` values of the left-hand side, while the former will not. Precedence rules can make this confusing. For example, in `false, 1 // 2` the left-hand side of `//` is `1`, not `false, 1` – `false, 1 // 2` parses the same way as `false, (1 // 2)`. In `(false, null, 1) | . // 42` the left-hand side of `//` is `.`, which always produces just one value, while in `(false, null, 1) // 42` the left-hand side is a generator of three values, and since it produces a value other `false` and `null`, the default `42` is not produced.

Examples

Filter	`empty // 42`
Input	`null`
Output	`42`
Run	

Filter	`.foo // 42`
Input	`{"foo": 19}`
Output	`19`

Run	

Filter	`.foo // 42"`
Input	`{}`
Output	`42`
Run	

Filter	`(false, null, 1) // 42`
Input	`null`
Output	`1`
Run	

Filter	`(false, null, 1) | . // 42`
Input	`null`
Output	`42`
	`42`
	`1`
Run	

try-catch

Errors can be caught by using `try EXP catch EXP`. The first expression is executed, and if it fails then the second is executed with the error message. The output of the handler, if any, is output as if it had been the output of the expression to try.

The `try EXP` form uses `empty` as the exception handler.

Examples

Filter	`try .a catch ". is not an object"`
Input	`true`
Output	`". is not an object"`
Run	

Filter	`[.[]|try .a]`
Input	`[{}, true, {"a":1}]`
Output	`[null, 1]`
Run	

Filter	`try error("some exception") catch .`
Input	`true`
Output	`"some exception"`

Run

Breaking out of control structures

A convenient use of try/catch is to break out of control structures like `reduce`, `foreach`, `while`, and so on.

For example:

```
# Repeat an expression until it raises "break" as an
# error, then stop repeating without re-raising the error.
# But if the error caught is not "break" then re-raise it.
try repeat(exp) catch if .=="break" then empty else error
```

jq has a syntax for named lexical labels to “break” or “go (back) to”:

```
label $out | ... break $out ...
```

The `break $label_name` expression will cause the program to act as though the nearest (to the left) `label $label_name` produced `empty`.

The relationship between the `break` and corresponding `label` is lexical: the label has to be “visible” from the break.

To break out of a `reduce`, for example:

```
label $out | reduce .[] as $item (null; if .==false then break $out
else ... end)
```

The following jq program produces a syntax error:

```
break $out
```

because no label `$out` is visible.

Error Suppression / Optional Operator: `?`

The `?` operator, used as `EXP?`, is shorthand for `try EXP`.

```
[.[] | .a?]
[{}, true, {"a":1}]
[null, 1]
```

```
[.[] | tonumber?]
["1", "invalid", "3", 4]
[1, 3, 4]
```

I/O

At this time jq has minimal support for I/O, mostly in the form of control over when inputs are read. Two builtins functions are provided for this, `input` and `inputs`, that read from the same sources (e.g., `stdin`, files named on the command-line) as jq itself. These two builtins, and jq’s own reading actions, can be interleaved with each other. They are commonly used in combination with the null input option `-n` to prevent one input from being read implicitly.

Two builtins provide minimal output capabilities, `debug`, and `stderr`. (Recall that a jq program’s output values are always output as JSON texts on `stdout`.) The `debug` builtin can have application-specific behavior, such as for executables that use the libjq C API but aren’t the jq executable itself. The `stderr` builtin outputs its input in raw mode to stder with no additional decoration, not even a newline.

Most jq builtins are referentially transparent, and yield constant and repeatable value streams when applied to constant inputs. This is not true of I/O builtins.

`input`

Outputs one new input.

Note that when using `input` it is generally be necessary to invoke jq with the `-n` command-line option, otherwise the first entity will be lost.

```
echo 1 2 3 4 | jq '[., input]' # [1,2] [3,4]
```

`inputs`

Outputs all remaining inputs, one by one.

This is primarily useful for reductions over a program’s inputs. Note that when using `inputs` it is generally necessary to invoke jq with the `-n` command-line option, otherwise the first entity will be lost.

```
echo 1 2 3 | jq -n 'reduce inputs as $i (0; . + $i)' # 6
```

`debug, debug(msgs)`

These two filters are like `.` but have as a side-effect the production of one or more messages on stderr.

The message produced by the `debug` filter has the form

```
["DEBUG:",<input-value>]
```

where `<input-value>` is a compact rendition of the input value. This format may change in the future.

The `debug(msgs)` filter is defined as `(msgs | debug | empty), .` thus allowing great flexibility in the content of the message, while also allowing multi-line debugging statements to be created.

For example, the expression:

```
1 as $x | 2 | debug("Entering function foo with $x == \($x)", .) | (.
+1)
```

would produce the value 3 but with the following two lines being written to stderr:

```
["DEBUG:","Entering function foo with $x == 1"]
["DEBUG:",2]
```

`stderr`

Prints its input in raw and compact mode to stderr with no additional decoration, not even a newline.

`input_filename`

Returns the name of the file whose input is currently being filtered. Note that this will not work well unless jq is running in a UTF-8 locale.

`input_line_number`

Returns the line number of the input currently being filtered.