jq is a tool to transform JSON data in various ways, such as selecting, iterating, and reducing. For instance, running the command `jq 'map(.price) | add'` takes an array of JSON objects as input and returns the sum of their "price" fields.

A jq program, such as `map(.price) | add`, is called a *filter*. Each filter takes an input value and produces a *stream* of output values. For instance, when the input value is an array, the filter `.[]` yields all the elements of the array. Even literals like `"hello"` or `42` are filters — they take an input and produce the same literal as output.

The simplest filter (or jq program) is identity `.`, which simply outputs its input. Because the default behavior of jq is to pretty-print all outputs, you can use `jq '.'` to validate and pretty-print JSON input. However, the jq programming language is quite rich and allows for much more than just validation and pretty-printing.

There are many filters for various standard tasks, such as extracting a particular field of an object, or converting a number to a string.

Filters can be combined in various ways. For example, you can feed the output of one filter to another filter, or collect the output of a filter into an array. Generally, things that would be done with loops and iteration in other languages are just done by gluing filters together in jq.

We can run a filter `FILTER` using `jq FILTER`, e.g. `jq .foo`. For large filters, it may be more convenient to write it into some `FILE` and to run it with `jq -f FILE`, e.g. `jq -f filter.jq`.

Note

> When using `jq FILTER`, it is important to mind the shell's quoting rules. As a general rule, it's best to always quote the jq program, because many characters with special meaning to jq are also shell meta-characters. For example, `jq "foo"` will fail on most Unix shells because that will be the same as `jq foo`, which will generally fail because `foo is not defined`.
>
> The quoting rules depend on your shell: When using a Unix shell, use single quotes around your jq program, When using the Windows command shell (`cmd.exe`), use *double quotes* around your jq program and escape double quotes in the jq program with backslashes. When using the Powershell (`powershell.exe`) or the Powershell Core (`pwsh`/`pwsh.exe`), use *single quotes* around your jq program and escape double-quotes in the jq program with backslashes (`\"`).
>
> - Unix shells: `jq '.["foo"]'`
> - Powershell: `jq '.[\"foo\"]'`
> - Windows command shell: `jq ".[\"foo\"]"`

By default, jq reads a stream of JSON values (including numbers and other literals) from a list of files (or `stdin` if no files are given), Whitespace is only needed to separate numbers (such as 1 and 2) and booleans (true and false). Using `--raw-input`, jq accepts arbitrary text as input. jq runs the given filter on each input value, and writes all output values of the filter to standard output, as a sequence of newline-separated JSON values.

Command-line options

Compatibility

> There exist several compilers/interpreters for the jq language; the reference implementation is called jq, but there is also gojq and jaq. This manual tries to point out when these implementations diverge from the reference implementation.

You can affect how jq reads and writes its input and output using some command-line options:

General options

- `-f filename` / `--from-file filename`:

  Read filter from the file rather than from a command line, like awk's `-f` option.

- `-L directory`:

  Prepend `directory` to the search paths for modules. If this option is used then no builtin search paths are used.

- `--exit-status` / `-e`:

  Sets the exit status of jq to 0 if the last output value was neither `false` nor `null`, 1 if the last output value was either `false` or `null`, or 4 if no valid result was ever produced. Normally jq exits with 2 if there was any usage problem or system error, 3 if there was a jq program compile error, or 0 if the jq program ran.

  You can also set the exit status with the `halt_error` function.

- `--version` / `-V`:

  Output the jq version and exit with zero.

- `--help` / `-h`:

  Output the jq help and exit with zero.

- `--`:

  Terminates argument processing. Remaining arguments are not interpreted as options.

Input options

- `--null-input` / `-n`:

  Don't read any input at all. Instead, the filter is run once using `null` as the input. This is useful when using jq as a simple calculator or to construct JSON data from scratch.

- `--raw-input` / `-R`:

  Don't parse the input as JSON. Instead, each line of text is passed to the filter as a string. If combined with `--slurp`, then the entire input is passed to the filter as a single long string.

- `--slurp` / `-s`:

  Instead of running the filter for each JSON object in the input, read the entire input stream into a large array and run the filter just once.

  Compatibility

  > When this option is used, jq combines the inputs of all files into one single array, whereas jaq yields an array for every file. This is motivated by jaq's `-i` / `--in-place` option, which could not work with the slurping behaviour implemented by jq. The behaviour of jq can be approximated in jaq; for example, to achieve the output of `jq -s . a b`, you may use `jaq -s . <(cat a b)`.

- `--stream`:

  Parse the input in streaming fashion, outputting arrays of path and leaf values (scalars and empty arrays or empty objects). For example, `"a"` becomes `[[],"a"]`, and `[[],"a",["b"]]` becomes `[[0],[]]`, `[[1],"a"]`, and `[[2,0],"b"]`.

This is useful for processing large inputs incrementally, in particular in conjunction with filtering and the `reduce` and `foreach` filters.

Several builtin functions help processing streaming input.

Compatibility

> jaq does not support this option.

- `--stream-errors`:

  Like `--stream`, but invalid JSON inputs yield array values where the first element is the error and the second is a path. For example, `["a",n]` produces `["Invalid literal at line 1, column 7", [1]]`.

  Implies `--stream`. Invalid JSON inputs produce no error values when `--stream` without `--stream-errors`.

  Compatibility

  > jaq does not support this option.

Output options

- `--compact-output` / `-c`:

  By default, jq pretty-prints JSON output. Using this option will result in more compact output by instead putting each JSON object on a single line.

- `--raw-output` / `-r`:

  With this option, if the filter's result is a string then it will be written directly to standard output rather than being formatted as a JSON string with quotes. This can be useful for making jq filters talk to non-JSON-based systems.

- `--raw-output0`:

  Like `-r` but jq will print NUL instead of newline after each output. This can be useful when the values being output can contain newlines. When an output value contains NUL, jq exits with non-zero code.

  Compatibility

  > jaq does not support this option.

- `--join-output` / `-j`:

  Like `-r` but jq won't print a newline after each output.

  Compatibility

  > In jaq, this does not enable `--raw-output`.

- `--ascii-output` / `-a`:

  jq usually outputs non-ASCII Unicode codepoints as UTF-8, even if the input specified them as escape sequences (like "3bc"). Using this option, you can force jq to produce pure ASCII output with every non-ASCII character replaced with the equivalent escape sequence.

  Compatibility

  > jaq does not support this option.

- `--sort-keys` / `-S`:

Output the fields of each object with the keys in sorted order.

Compatibility

> jaq does not support this option. However, you can sort all output objects by their keys using a filter such as:
>
> ```
> walk(if . >= {} then reduce (keys[] as $k | { ($k): .[$k] }) as $o ({}; . + $o)
> end)
> ```

- `--color-output` / `-C` and `--monochrome-output` / `-M`:

  By default, jq outputs colored JSON if writing to a terminal. You can force it to produce color even if writing to a pipe or a file using `-C`, and disable color with `-M`. When the `NO_COLOR` environment variable is not empty, jq disables colored output by default, but you can enable it by `-C`.

  Colors can be configured with the `JQ_COLORS` environment variable.

  Compatibility

  > In jaq, the corresponding options are called `--color=always` and `--color=never`.

- `--tab`:

  Use a tab for each indentation level instead of two spaces.

- `--indent n`:

  Use the given number of spaces (no more than 7) for indentation.

- `--unbuffered`:

  Flush the output after each JSON object is printed. This is useful if you pipe a slow data source into jq and pipe jq's output elsewhere.

  Compatibility

  > jaq does not support this option.

- `--seq`:

  Use the `application/json-seq` MIME type scheme for separating JSON texts in jq's input and output. This means that an ASCII RS (record separator) character is printed before each value on output and an ASCII LF (line feed) is printed after every output. Input JSON texts that fail to parse are ignored (but warned about), discarding all subsequent input until the next RS. This mode also parses the output of jq without the `--seq` option.

  Compatibility

  > jaq does not support this option.

- `--binary` / `-b`:

  Windows users using WSL, MSYS2, or Cygwin, should use this option when using a native jq.exe, otherwise jq will turn newlines (LFs) into carriage-return-then-newline (CRLF).

  Compatibility

  > jaq does not support this option.

Variable options

- `--arg name value`:

This option passes a value to the jq program as a predefined variable. If you run jq with `--arg foo bar`, then `$foo` is available in the program and has the value `"bar"`. Note that `value` will be treated as a string, so `--arg foo 123` will bind `$foo` to `"123"`.

Named arguments are also available to the jq program as `$ARGS.named`.

- `--argjson name JSON-text`:

  This option passes a JSON-encoded value to the jq program as a predefined variable. If you run jq with `--argjson foo 123`, then `$foo` is available in the program and has the value `123`.

- `--slurpfile variable-name filename`:

  This option reads all the JSON texts in the named file and binds an array of the parsed JSON values to the given global variable. If you run jq with `--slurpfile foo bar`, then `$foo` is available in the program and has an array whose elements correspond to the texts in the file named `bar`.

- `--rawfile variable-name filename`:

  This option reads in the named file and binds its contents to the given global variable. If you run jq with `--rawfile foo bar`, then `$foo` is available in the program and has a string whose contents are to the texts in the file named `bar`.

- `--args`:

  Remaining arguments are positional string arguments. These are available to the jq program as `$ARGS.positional[]`.

  Compatibility

  jaq does not support this option.

- `--jsonargs`:

  Remaining arguments are positional JSON text arguments. These are available to the jq program as `$ARGS.positional[]`.

  Compatibility

  jaq does not support this option.

Development options

- `--build-configuration`:

  Output the build configuration of jq and exit with zero. This output has no supported format or structure and may change without notice in future releases.

  Compatibility

  jaq does not support this option.

- `--run-tests [filename]`:

  Runs the tests in the given file or standard input. This must be the last option given and does not honor all preceding options. The input consists of comment lines, empty lines, and program lines followed by one input line, as many lines of output as are expected (one per output), and a terminating empty line. Compilation failure tests start with a line containing only `%%FAIL`, then a line containing the program to compile, then a line containing an error message to compare to the actual.

  This option can change backwards-incompatibly.

Colors

To configure alternative colors, you may set the `JQ_COLORS` environment variable to colon-delimited list of partial terminal escape sequences like `"1;31"`, in this order:

- color for `null`
- color for `false`
- color for `true`
- color for numbers
- color for strings
- color for arrays
- color for objects
- color for object keys

The default color scheme is the same as setting `JQ_COLORS="0;90:0;39:0;39:0;39:0;32:1;39:1;39:1;34"`.

This is not a manual for VT100/ANSI escapes. However, each of these color specifications should consist of two numbers separated by a semi-colon. The first number is one of these:

- 1 (bright)
- 2 (dim)
- 4 (underscore)
- 5 (blink)
- 7 (reverse)
- 8 (hidden)

The second number is one of these:

- 30 (black)
- 31 (red)
- 32 (green)
- 33 (yellow)
- 34 (blue)
- 35 (magenta)
- 36 (cyan)
- 37 (white)

Compatibility

jaq does not consider `JQ_COLORS`.

Types and Values

jq supports the same set of datatypes as JSON — booleans, numbers, strings, arrays, objects (JSON-speak for hashes with only string keys), and `null`. This section covers how to create values in jq.

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

Booleans

The booleans can be produced by the filters `true` and `false`.

Numbers

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 stores the original literal string. When a number which originally was provided as a literal is never mutated until the end of the program, then its original literal form is preserved. This also includes cases when the original literal would be truncated when converted to an IEEE754 double precision floating point number.

Note

> Using the current implementation of jq, the expression `1E1234567890` produces `1.7976931348623157e+308` on at least one platform. This is because, in the process of parsing the number, jq has converted it to an IEEE754 double-precision representation, losing precision.
>
> The way in which jq handles numbers has changed over time. Further changes are likely within the parameters set by the relevant JSON standards. Moreover, build configuration options can alter how jq processes numbers.
>
> The following remarks are therefore offered with the understanding that they are intended to be descriptive of the current version of jq and should not be interpreted as being prescriptive:
>
> (1) Any arithmetic operation on a number that has not already been converted to an IEEE754 double precision representation will trigger a conversion to the IEEE754 representation.
>
> (2) jq will attempt to maintain the original decimal precision of number literals (unless the `--disable-decnum` build configuration option was used), but in expressions such as `1E1234567890`, precision will be lost if the exponent is too large.
>
> (3) In jq programs, a leading minus sign triggers the conversion of the number to an IEEE754 representation.
>
> (4) Comparisons are carried out using the untruncated big decimal representation of numbers if available, as illustrated in one of the following examples.
>
> See the builtin function `have_decnum` for examples where the `--disable-decnum` build configuration option matters.

Compatibility

> In gojq and jaq, numbers are either floating-point numbers or integers.

Examples

Filter	`. < 0.12345678901234567890123456788`
Input	`0.12345678901234567890123456789`
Output	`false`
Run	

Strings

String interpolation: `\(f)`

Inside a string, you can put a filter inside parentheses after a backslash, such as:

```
"Hello \(.name) of planet \(.planet)!"
```

The output of the filter will be interpolated into the string. The example above is equivalent to:

```
"Hello " + (.name) + " of planet " + (.planet) + "!"
```

Examples

Filter	`"The input was \(.), which is one less than \(.+1)"`
Input	`42`
Output	`"The input was 42, which is one less than 43"`
Run	

String formatting: `@f`

The `@foo` syntax is used to format and escape strings, which is useful for building URLs, documents in a language like HTML or XML, and so forth. `@foo` can also be used as a filter on its own. See below for a list.

This syntax can be combined with string interpolation in a useful way. You can follow a `@foo` token with a string literal. The contents of the string literal will *not* be escaped. However, all interpolations made inside that string literal will be escaped. For instance,

```
@uri "https://www.google.com/search?q=\(.search)"
```

will produce the following output for the input `{"search":"what is jq?"}`:

```
"https://www.google.com/search?q=what%20is%20jq%3F"
```

Note that the slashes, question mark, etc. in the URL are not escaped, as they were part of the string literal.

Examples

Filter	`@html`
Input	`"This works if x < y"`
Output	`"This works if x &lt; y"`
Run	

Filter	`@sh "echo \(.)"`
Input	`"O'Hara's Ale"`
Output	`"echo 'O'\\''Hara'\\''s Ale'"`
Run	

Filter	`@base64`
Input	`"This is a message"`
Output	`"VGhpcyBpcyBhIG1lc3NhZ2U="`
Run	

Filter	`@base64d`
Input	`"VGhpcyBpcyBhIG1lc3NhZ2U="`
Output	`"This is a message"`
Run	

Arrays: `[]`

As in JSON, `[]` is used to construct arrays, as in `[1,2,3]`. The elements of the arrays can be any jq expression. 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 concatenation 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 `f` that produces four results, then the expression `[f]` 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	

Objects: `{}`

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 need to wrap it in parentheses if it contains colons), which gets applied to the `{}` expression's input (remember, all filters have an input and an output). For example,

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

produces 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	

Basic filters

Identity: `.`

The simplest filter is `.` and is called the *identity operator*. This filter takes its input and produces the same value as output.

Since jq by default pretty-prints all output, a trivial program consisting of nothing but `.` can be used to format JSON output from, say, `curl`.

Examples

Filter	`.`
Input	`"Hello, world!"`
Output	`"Hello, world!"`
Run	

Filter	`.`
Input	`0.12345678901234567890123456789`
Output	`0.12345678901234567890123456789`
Run	

Concatenation: `,`

The `,` operator concatenates the outputs of two filters.

Given two filters `f` and `g`, their concatenation `f, g` first returns the outputs of `f`, then the outputs of `g`. The input of `f, g` is fed to both `f` and `g`.

Example

The filter `.foo, .bar` produces both the “foo” fields and “bar” fields as separate outputs.

Examples

Filter	`.foo, .bar`
Input	`{"foo": 42, "bar": "something else", "baz": true}`

Output	`42`
	`"something else"`

Run	

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

Output	`"stedolan"`
	`"jq"`
	`"wikiflow"`

Run	

Filter	`.[4,2]`
Input	`["a","b","c","d","e"]`
Output	`"e"` `"c"`
Run	

Composition: `|`

The `|` operator feeds the output of one filter to another filter. It’s similar to the Unix shell’s pipe, if you’re used to that.

Given two filters `f` and `g`, their composition `f | g` feeds the input of `f | g` to `f`, and for every output of `f`, feeds it to `g` and returns its outputs.

Example

The expression `.[] | .foo` retrieves the “foo” field of each element of the input array.

Note too that `.` is the input value at the particular stage in a “pipeline”, specifically: where the `.` expression appears. Thus `.a | . | .b` is the same as `.a.b`, as the `.` in the middle refers to whatever value `.a` produced.

Examples

Filter	`.[] | .name`
Input	`[{"name":"JSON", "good":true}, {"name":"XML", "good":false}]`

Output	`"JSON"`
	 `"XML"`

Run	

Function call

jq provides many builtin functions for a variety of tasks, and you can also define your own functions.

Each function has an *arity* that specifies how many *arguments* that function takes. For example, the function `length` does not take any argument, so its arity is 0, whereas the function `contains` takes one argument, so its arity is 1. There may be several functions that have the same name, but different arities; for example, there exist two functions called `add`, taking zero and one arguments, respectively. To unambiguously identify a function `f` with arity `n`, we write `f/n`, e.g. `length/0`, `contains/1`, `add/0`, and `add/1`.

To call a function `f` with no arguments (arity 0), we write `f`. To call a function `f` with `n` arguments (arity greater than zero), we write `f(a1; ...; an)`.

Note

Function calls use semicolons `;` instead of commas `,` to separate arguments, because `,` is already used for concatenation.

Examples

The first example calls the function `length/0`:

Filter	`length`
Input	`[1, 1, 2, 3]`
Output	`4`
Run	

The next example calls `range/2` and `add/1`:

Filter	`add(range(0; .))`
Input	`2`
Output	`3`
Run	

The last example calls `while/2`:

Filter	`while(length < 3; . + "a")`
Input	`""`

Output	`""`
	 `"a"`
	 `"aa"`

Run	

Parenthesis

Parentheses act as a grouping operator just as in any typical programming language.

Examples

Filter	`(. + 2) * 5`
Input	`1`
Output	`15`
Run	

Recursive descent: `..`

Recursively descends `.`, producing every value. This is the same as the zero-argument `recurse` function. 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)` and the `?` operator.

Examples

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

Paths

In this section, we will show three very frequently used operators, namely for iteration, indexing, and slicing. These operators serve to obtain parts of values. Furthermore, we will see how to combine these operators.

Iteration operator: `.[]`

The operator `.[]` returns the values contained inside the input value.

If the input is an array, then `.[]` returns all elements of the array, and if the input is an object, `.[]` returns all the values of the object. For example, running `.[]` with the input `[1,2,3]` produces the numbers `1 2 3` as three separate results, rather than as a single array.

Examples

Filter	`.[]`
Input	`[{"name":"JSON", "good":true}, {"name":"XML", "good":false}]`

Output	`{"name":"JSON", "good":true}`
	`{"name":"XML", "good":false}`

Run	

Filter	`.[]`
Input	`[]`
Output	
Run	

Filter	`.foo[]`
Input	`{"foo":[1,2,3]}`

Output	`1`
	`2`
	`3`

Run	

Filter	`.[]`
Input	`{"a": 1, "b": 1}`
Output	`1` `1`
Run	

Indexing operator: `.[f]`

When given an array as input, `.[n]` produces the `n`-th element of the array. For example, given the array `[2, 4, 6]`, the filter `.[1]` returns `4`. Arrays are zero-based, so `.[2]` returns the third element. Negative indices are allowed, with −1 referring to the last element, −2 referring to the next to last element, and so on.

When given a JSON object as input, `.[k]` produces the value at the key `k` if it is present in the object, or `null` otherwise. For example, given the object `{name: "Anna", age: 24}`, the filter `.["name"]` produces `"Anna"`, `.["age"]` produces `24`, and `.["address"]` produces `null`.

We say that a key is *identifier-like* when it does not start with a digit and consists only of alphanumeric characters and underscores; for example, `"foo"` is identifier-like. For identifier-like keys like `"foo"`, you can also look up the field `"foo"` of an object using the shorthand syntax `.foo`. For example, we could have written `.name`, `.age`, and `.address` above, whereas we cannot use this shorthand syntax for `.["foo::bar"]` and `.["foo.bar"]`.

Compatibility

> In jq, when given `null` input, `.["a"]` and `.[0]` yield `null`, but `.[]` yields an error. jaq yields an error in all cases to prevent accidental indexing of `null` values. To obtain the same behaviour in jq and jaq, you can use `.["a"]? // null` or `.[0]? // null` instead.

Examples

Filter	`.foo`
Input	`{"foo": 42, "bar": "less interesting data"}`
Output	`42`
Run	

Filter	`.foo`
Input	`{"notfoo": true, "alsonotfoo": false}`
Output	`null`
Run	

Filter	`.["foo"]`
Input	`{"foo": 42}`
Output	`42`
Run	

Filter	`.[0]`
Input	`[{"name":"JSON", "good":true}, {"name":"XML", "good":false}]`
Output	`{"name":"JSON", "good":true}`
Run	

Filter	`.[2]`
Input	`[{"name":"JSON", "good":true}, {"name":"XML", "good":false}]`
Output	`null`
Run	

Filter	`.[-2]`
Input	`[1,2,3]`
Output	`2`
Run	

Slicing operator: `.[f:g]`

The operator `.[f:g]` returns a slice of an array or a string. For example, when given an array, `.[10:15]` returns an array of length 5, containing the elements from index 10 (inclusive) to index 15 (exclusive). Either index may be negative, in which case it counts backwards from the end of the array. If `f` is omitted, it is assumed to be `0`, and if `g` is omitted, it is assumed to be `length`. Indices are zero-based.

Examples

Filter	`.[2:4]`
Input	`["a","b","c","d","e"]`
Output	`["c", "d"]`
Run	

Filter	`.[2:4]`
Input	`"abcdefghi"`
Output	`"cd"`
Run	

Filter	`.[:3]`
Input	`["a","b","c","d","e"]`
Output	`["a", "b", "c"]`
Run	

Filter	`.[-2:]`
Input	`["a","b","c","d","e"]`
Output	`["d", "e"]`
Run	

Compound paths

Frequently, when using the path operators given above, we find ourselves combining them with the `|` and `?` operators. Therefore, jq provides shorthand syntax for these combinations. For example:

- `.key[]` for `.key | .[]`,
- `.[].key?` for `.[] | .key?`,
- `.[]?[]` for `.[]? | .[]`,
- `.a.b` for `.a | .b`,
- `.a.b.c` for `.a | .b | .c`, and so on.

We call such a combination a *compound path*.

The rules for what constitutes a compound path are surprisingly complex. Therefore, we define it via a formal grammar in EBNF:

```
path = atomic, part
     | ".", ident
     | path, part
     | path, part, "?"
     ;

part = ".", ident
     | "[",                "]"
     | "[",      t,        "]"
     | "[", t, ":", t, "]"
     | "[", t, ":",        "]"
     | "[",      ":", t, "]"
     ;
```

Here, `ident` refers to an identifier-like key, `t` refers to a filter, and `atomic` refers to an *atomic* filter, such as `.` (identity), function call, and parenthesis. This grammar defines a compound `path` as a sequence of path parts, potentially prefixed by an atomic root. A path `part` is any of the operators previously introduced in this section. (Note that `part` does not include the leading `.` for all operators except for `.ident`.)

Example

The following is a compound path:

```
add[].posts[0]?.sections[]["title"]?
```

We can decompose it into its different parts:

```
add           # atomic (function call)
[]            # iteration
.posts        # indexing
[0]?          # indexing
.sections     # indexing
[]            # iteration
["title"]?    # indexing
```

We can transform this into an equivalent filter:

```
  add
| .[]
| .posts
| .[0]?
| .sections
| .[]
| .["title"]?
```

Filters inside a part of a compound path, such as `f` and `g` in `.[f][:g]`, are run with the *input given to the whole path.*

Example

When we run the filter `.arr[][.key]` on the input `{key: "a", arr: [{a: 1, b: 2}, {a: 3}]}`, then `.key` is run on the original input, not on the current value returned by `.arr[]`! To see the difference, let us first consider a **wrong** transformation:

```
  .arr          # --> [{a: 1, b: 2}, {a: 3}]
| .[]           # -->  {a: 1, b: 2}, {a: 3}
| .[.key]       # -->  error  (because .key is run with input {a: 1, b: 2} and
yields null)
```

Now, let us consider a **correct** transformation:

```
  .key as $x    # --> "a"
| .arr          # --> [{a: 1, b: 2}, {a: 3}]
| .[]           # -->  {a: 1, b: 2}, {a: 3}
| .[$x]         # -->  1, 3
```

Note

Surprisingly, the filter `.[f]?` is **not** equivalent to `(.[f])?`. To see this, let us transform `.[f]?` to an equivalent filter like above:

```
  f as $x
| .[$x]?
```

The difference shows when `f` causes an error — in that case, `.[f]?` will raise the error, whereas `(.[f])?` will not raise *any* error.

Examples

Filter	`.foo?`
Input	`{"foo": 42, "bar": "less interesting data"}`
Output	`42`
Run	

Filter	`.foo?`
Input	`{"notfoo": true, "alsonotfoo": false}`
Output	`null`
Run	

Filter	`.["foo"]?`
Input	`{"foo": 42}`
Output	`42`
Run	

Filter	`[.foo?]`
Input	`[1,2]`
Output	`[]`
Run	

Arithmetic and Comparison

We are now going to introduce operators for arithmetic (`+`, `-` `*`, `/`, `%`), equality (`==`, `!=`), and ordering (`<`, `<=`, `>`, `>=`).

All operators in this section feed their input to both arguments and combine their results. This allows us to implement an averaging filter as `add / length` — this feeds the input both to the `add` filter and the `length` filter, then performs the division of their results.

Given two filters `f` and `g`, we can write `f + g`, `f == g`, `f < g` and so on to perform the desired operation on the outputs of the filters `f` and `g`. When `f` or `g` outputs multiple values, then all combinations of the operation are performed.

Example

> Suppose that `f` outputs the values `1, 2` and `g` outputs the values `3, 4`. Then `f * g` outputs the values `3, 6, 4, 8`.

Compatibility

> For any operator in this section such as `+`, in jq , `f + g` is equivalent to `g as $y | f as $x | $x + $y`, whereas in jaq, `f + g` is equivalent to `f as $x | g as $y | $x + $y`. That means that in the above example, jaq outputs the values `3, 4, 6, 8` instead of `3, 6, 4, 8`. Note that this difference shows only when both `f` and `g` produce multiple values.

Some jq operators (for instance, `+`) do different things depending on the type of their arguments (arrays, numbers, etc.). However, jq never does implicit type conversions. Trying to add a string to an object results in an error.

Addition: `+`

The operator `+` takes two filters, applies them both to the same input, and adds the results together. What “adding” means depends on the types involved:

- **Numbers** are added by normal arithmetic.
- **Arrays** are added by being concatenated into a larger array.
- **Strings** are added by being joined into a larger string.
- **Objects** are added by merging, that is, inserting all the key-value pairs from both objects into a single combined object. If both objects contain a value for the same key, the object on the right of the `+` wins. (For recursive merge use the `*` operator.)

`null` can be added to any value, and returns the other value unchanged.

Examples

Filter	`.a + 1`
Input	`{"a": 7}`
Output	`8`
Run	

Filter	`.a + .b`
Input	`{"a": [1,2], "b": [3,4]}`
Output	`[1,2,3,4]`
Run	

Filter	`.a + null`
Input	`{"a": 1}`
Output	`1`
Run	

Filter	`.a + 1`
Input	`{}`
Output	`1`
Run	

Filter	`{a: 1} + {b: 2} + {c: 3} + {a: 42}`
Input	`null`
Output	`{"a": 42, "b": 2, "c": 3}`
Run	

Subtraction: `-`

As well as normal arithmetic subtraction on numbers, the `-` operator can be used on arrays to remove all occurrences of the second array's elements from the first array.

Examples

Filter	`4 - .a`
Input	`{"a":3}`
Output	`1`
Run	

Filter	`. - ["xml", "yaml"]`
Input	`["xml", "yaml", "json"]`
Output	`["json"]`
Run	

Multiplication, division, modulo: `*`, `/`, `%`

These infix operators behave as expected when given two numbers. Division by zero raises an error. `x % y` computes x modulo y.

Multiplying a string by a number produces the concatenation of that string that many times. `"x" * 0` produces `""`.

Dividing a string by another splits the first using the second as separators.

Multiplying two objects will merge them recursively: this works like addition but if both objects contain a value for the same key, and the values are objects, the two are merged with the same strategy.

Examples

Filter	`10 / . * 3`
Input	`5`
Output	`6`
Run	

Filter	`. / ", "`
Input	`"a, b,c,d, e"`
Output	`["a","b,c,d","e"]`
Run	

Filter	`{"k": {"a": 1, "b": 2}} * {"k": {"a": 0,"c": 3}}`
Input	`null`
Output	`{"k": {"a": 0, "b": 2, "c": 3}}`
Run	

Filter	`.[] | (1 / .)?`
Input	`[1,0,-1]`

Output	`1`
	`-1`

Run	

Equality: ==, !=

The expression `a == b` produces `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.

The expression `a != b` returns `false` if `a == b` returns `true`, else `true`.

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	

Ordering: >, >=, <=, <

The ordering 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).

Values are ordered as follows, by increasing order:

- `null`
- `false`
- `true`
- numbers
- strings, in alphabetical order (by unicode codepoint value)
- arrays, in lexical order
- objects

The ordering for objects is a little complex: first they're compared by comparing their sets of keys (as arrays in sorted order), and if their keys are equal then the values are compared key by key.

Examples

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

Boolean filters

Every value can be converted to a boolean — in particular, the values `false` and `null` have the *boolean value* `false`, all other values have the boolean value `true`. This section describes several filters that analyze the boolean value of values.

You can negate the boolean value of a value with the builtin function `not`. It is called as a filter to which things can be piped rather than with special syntax, as in `.foo and .bar | not`.

if-then-else-end

The filter `if i then t else e end` runs `t` when `i` returns an output with boolean value `true`, otherwise, it runs `e`.

Given three filters `i`, `t`, and `e`, the expression `if i then t else e end` runs `i` on its input. For every value `y` that is output by `i`, if `y` has the boolean value `true` (that means, if `y` is neither `false` nor `null`), the output of `t` on the original input is produced, else the output of `e` on the original input is produced.

Note

> 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.

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

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

Examples

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

Filter	`.[] | if . then ., .+1 else . end`
Input	`[false, 1, null]`

Output	`false`
	`1`
	`2`
	`null`

Run	

`and`, `or`

The filter `f and g` returns true if both `f` and `g` return an output with boolean value `true`. The filter `f or g` returns true if either `f` or `g` return an output with boolean value `true`. Otherwise, both filters return `false`.

Given two filters `f` and `g`, their conjunction `f and g` and their disjunction `f or g` run `f` on the input, and for every output `y` of `f`, they analyze the boolean value `y` of the output:

- `f and g` yields `false` if `y` is `false`, otherwise it runs `g` with the original input and yields the boolean values of its outputs.
- `f or g` yields `true` if `y` is `true`, otherwise it runs `g` with the original input and yields the boolean values of its outputs.

Example

> The filter `true and false` returns `false`, whereas `true or false` returns `true`.

Note

> These filters only produce the values `true` and `false`, 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: `//`

Given two filters `f` and `g`, the filter `f // g` runs `f` on the input and yields all of its outputs whose boolean value is `true`. If the boolean values of all outputs of `f` are `false` (which is also the case if `f` does not yield any output at all), then `f // g` runs `g` with the original input and yields its outputs.

This is useful for providing defaults: `.foo // 1` evaluates 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).

Note

> `f // g` is not the same as `f | (. // g)` (which can be written more compactly as `f | . // g`). The latter produces default values for *all* outputs of `f` whose boolean value is `false`, while the former does not.

Example

> The filter `(false, null, 1) | . // 42` yields the outputs `42, 42, 1`, whereas the filter `(false, null, 1) // 42` yields just `1`.

Note

> Mind the precedence rules. For example, in `false, 1 // 2` the left-hand side of `//` is `1`, not `false, 1`. This is because `false, 1 // 2` parses the same way as `false, (1 // 2)`.

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	

Error handling

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	

Error suppression: ?

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

Examples

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

Filter	`[.[] | tonumber?]`
Input	`["1", "invalid", "3", 4]`
Output	`[1, 3, 4]`
Run	

label-break

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.

Variables

Variables are an absolute necessity in most programming languages, but in jq, they can be considered an “advanced feature”.

In most languages, variables are the only means of passing around data. If you calculate a value, and you want to use it more than once, you’ll need to store it in a variable. To pass a value to another part of the program, you’ll need that part of the program to define a variable (as a function parameter, object member, or whatever) in which to place the data.

It is also possible to define functions in jq itself. In fact, many of jq’s built-in functions, including `map` and `select`, are written in jq.

Variable binding: `f as $x | g`

In jq, all filters have an input and an output, so manual plumbing is not necessary to pass a value from one part of a program to the next. Many expressions, for instance `a + b`, pass their input to two distinct subexpressions (here `a` and `b` are both passed the same input), so variables aren’t usually necessary in order to use a value twice.

For instance, calculating the average value of an array of numbers requires a few variables in most languages - at least one to hold the array, perhaps one for each element or for a loop counter. In jq, it’s simply `add / length` - the `add` expression is given the array and produces its sum, and the `length` expression is given the array and produces its length.

So, variables are often unnecessary and sometimes even best avoided, but jq does let you define variables using the syntax `f as $x`. All variable names start with `$`. Here’s a slightly uglier version of the array-averaging example:

```
length as $array_length | add / $array_length
```

We’ll need a more complicated problem to find a situation where using variables actually makes our lives easier.

Example

Suppose we have an array of blog posts, with “author” and “title” fields, and another object which is used to map author usernames to real names. Our input looks like:

```
{"posts": [{"title": "First post", "author": "anon"},
           {"title": "A well-written article", "author": "person1"}],
 "realnames": {"anon": "Anonymous Coward",
               "person1": "Person McPherson"}}
```

We want to produce the posts with the author field containing a real name, as in:

```
{"title": "First post", "author": "Anonymous Coward"}
{"title": "A well-written article", "author": "Person McPherson"}
```

We use a variable, `$names`, to store the realnames object, so that we can refer to it later when looking up author usernames:

```
.realnames as $names | .posts[] | {title, author: $names[.author]}
```

The filter `f as $x | g` runs `f` on its input, and for each output `y` produced by `f`, it runs `g` with the original input and with `$x` set to `y`. Thus `as` functions as something of a foreach loop.

Just as `{foo}` is a handy way of writing `{foo: .foo}`, so `{$foo}` is a handy way of writing `{foo: $foo}`.

Examples

Filter	`.bar as $x | .foo | . + $x`
Input	`{"foo":10, "bar":200}`
Output	`210`
Run	

Filter	`. as $i|[(.*2|. as $i| $i), $i]`
Input	`5`
Output	`[10,5]`
Run	

Scoping

There are three types of symbols in jq: variables, labels, and functions. All of these symbols are scoped lexically, with filters being able to refer only to symbols that have been defined "to the left" of them. Furthermore, there is no way to change the value of a binding; one can only create a new binding with the same name, but this will not be visible where the old one was.

Example

In the filter

```
.realnames as $names | (.posts[] | {title, author: $names[.author]})
```

the binding `$names` is visible "to the right" of it, but in the filter

```
(.realnames as $names | .posts[]) | {title, author: $names[.author]}
```

the binding `$names` is *not* visible past the closing parenthesis, so the filter is not well-formed.

Example

The filter `1 as $x | (2 as $x | $x), $x` returns the values `2, 1`. First, it introduces a variable `$x` via `1 as $x`, then it introduces a variable `$x` via `2 as $x` that *shadows* the previous `$x`. However, because we limit the scope of `2 as $x` with parentheses, the final `$x` refers to the original `1 as $x` again.

Note

Labels and variables look alike, yet they live in different worlds. To see this, consider the filter `1 as $x | label $x | $x, break $x, 2`. If the variable `$x` and the label `$x` would live in the same world, then the label `$x` would shadow the variable `$x`. However, because they live in different worlds, they do not shadow each other, therefore this filter is syntactically correct and returns `1`.

Destructuring

Multiple variables may be declared using a single `as` expression by providing a pattern that matches the structure of the input:

```
. as {realnames: $names, posts: [$first, $second]} | ...
```

The variable declarations in array patterns (e.g., `. as [$first, $second]`) bind the elements of the array from the element at index zero on up, in order. When there is no value at the index for an array pattern element, `null` is bound to that variable.

Compatibility

jaq does not support destructuring.

Examples

Filter	`. as [$a, $b, {c: $c}] | $a + $b + $c`
Input	`[2, 3, {"c": 4, "d": 5}]`
Output	`9`
Run	

Filter	`.[] as [$a, $b] | {a: $a, b: $b}`
Input	`[[0], [0, 1], [2, 1, 0]]`

Output	`{"a":0,"b":null}`
	`{"a":0,"b":1}`
	`{"a":2,"b":1}`

Run	

Filter	`foreach .[] as {("a", "b"): $x} ([]; . + [$x])`
Input	`[{"a": 1, "b": 2}, {"a": 3, "b": 4}]`

Output	`[1]`
	`[1,2]`
	`[1,2,3]`
	`[1,2,3,4]`

Run	

Destructuring alternative operator: `?//`

The destructuring alternative operator provides a concise mechanism for destructuring an input that can take one of several forms.

Suppose we have an API that returns a list of resources and events associated with them, and we want to get the user_id and timestamp of the first event for each resource. The API (having been clumsily converted from XML) will only wrap the events in an array if the resource has multiple events:

```
{"resources": [
  {"id": 1, "kind": "widget", "events": {"action": "create", "user_id": 1, "ts":
13}},
  {"id": 2, "kind": "widget", "events": [
    {"action": "create", "user_id": 1, "ts": 14},
    {"action": "destroy", "user_id": 1, "ts": 15}
  ]}
]}
```

We can use the destructuring alternative operator to handle this structural change simply:

```
.resources[] as {$id, $kind, events: {$user_id, $ts}} ?// {$id, $kind, events:
[{$user_id, $ts}]} |
{$user_id, $kind, $id, $ts}
```

Or, if we aren't sure if the input is an array of values or an object:

```
.[] as [$id, $kind, $user_id, $ts] ?// {$id, $kind, $user_id, $ts} | ...
```

Each alternative need not define all of the same variables, but all named variables will be available to the subsequent expression. Variables not matched in the alternative that succeeded will be `null`:

```
.resources[] as {$id, $kind, events: {$user_id, $ts}} ?// {$id, $kind, events:
[{$first_user_id, $first_ts}]} |
{$user_id, $first_user_id, $kind, $id, $ts, $first_ts}
```

Additionally, if the subsequent expression returns an error, the alternative operator will attempt to try the next binding. Errors that occur during the final alternative are passed through.

```
[[3]] | .[] as [$a] ?// [$b] | if $a != null then error("err: \($a)") else {$a,$b}
end
```

Compatibility

jaq does not support this operator.

Examples

Filter	`.[] as {$a, $b, c: {$d, $e}} ?// {$a, $b, c: [{$d, $e}]} \| {$a, $b, $d, $e}`
Input	`[{"a": 1, "b": 2, "c": {"d": 3, "e": 4}}, {"a": 1, "b": 2, "c": [{"d": 3, "e": 4}]}]`
Output	`{"a":1,"b":2,"d":3,"e":4}` `{"a":1,"b":2,"d":3,"e":4}`
Run	

Filter	`.[] as {$a, $b, c: {$d}} ?// {$a, $b, c: [{$e}]} | {$a, $b, $d, $e}`
Input	`[{"a": 1, "b": 2, "c": {"d": 3, "e": 4}}, {"a": 1, "b": 2, "c": [{"d": 3, "e": 4}]}]`

Output	`{"a":1,"b":2,"d":3,"e":null}`
	`{"a":1,"b":2,"d":null,"e":4}`

Run	

Filter	`.[] as [$a] ?// [$b] | if $a != null then error("err: \($a)") else {$a,$b} end`
Input	`[[3]]`
Output	`{"a":null,"b":3}`
Run	

Reduction

jq has reduction operators, which can be used to run a filter on every element of a stream while keeping some intermediate state. These operators are used to define some bits of jq's standard library, such as `add`.

`reduce`

The `reduce` syntax allows you to combine all of the results of an expression by accumulating them into a single answer. The form is `reduce EXP as $var (INIT; UPDATE)`. As an example, we'll pass `[1,2,3]` to this expression:

```
reduce .[] as $item (0; . + $item)
```

For each result that `.[]` produces, `. + $item` is run to accumulate a running total, starting from 0 as the input value. In this example, `.[]` produces the results `1`, `2`, and `3`, so the effect is similar to running something like this:

```
0 | 1 as $item | . + $item |
    2 as $item | . + $item |
    3 as $item | . + $item
```

Compatibility

When `UPDATE` yields multiple outputs, jq only considers the *last* one for the next iteration, whereas jaq considers *all* of them. For example, the filter

```
reduce (0, 1) as $x ([]; . + (["a", $x], ["b", $x]))
```

yields `["b",0,"b",1]` in jq, whereas in jaq, it yields:

```
["a",0,"a",1]
["a",0,"b",1]
["b",0,"a",1]
["b",0,"b",1]
```

Examples

Filter	`reduce .[] as $item (0; . + $item)`
Input	`[1,2,3,4,5]`
Output	`15`
Run	

Filter	`reduce .[] as [$i,$j] (0; . + $i * $j)`
Input	`[[1,2],[3,4],[5,6]]`
Output	`44`
Run	

Filter	`reduce .[] as {$x,$y} (null; .x += $x \| .y += [$y])`
Input	`[{"x":"a","y":1},{"x":"b","y":2},{"x":"c","y":3}]`
Output	`{"x":"abc","y":[1,2,3]}`
Run	

`foreach`

The `foreach` syntax is similar to `reduce`, but intended to allow the construction of `limit` and reducers that produce intermediate results.

The form is `foreach EXP as $var (INIT; UPDATE; EXTRACT)`. As an example, we'll pass `[1,2,3]` to this expression:

```
foreach .[] as $item (0; . + $item; [$item, . * 2])
```

Like the `reduce` syntax, `. + $item` is run for each result that `.[]` produces, but `[$item, . * 2]` is run for each intermediate values. In this example, since the intermediate values are `1`, `3`, and `6`, the `foreach` expression produces `[1,2]`, `[2,6]`, and `[3,12]`. So the effect is similar to running something like this:

```
0 | 1 as $item | . + $item | [$item, . * 2],
    2 as $item | . + $item | [$item, . * 2],
    3 as $item | . + $item | [$item, . * 2]
```

When `EXTRACT` is omitted, the identity filter is used. That is, it outputs the intermediate values as they are.

Compatibility

jaq does not provide `foreach/3`, but it does provide `foreach/2`.

Furthermore, similarly as for `reduce`, jq considers only the *last* output of `UPDATE` for the next iteration, whereas jq considers *all* of them. For example, the filter

```
foreach (0, 1) as $x ([]; . + (["a", $x], ["b", $x]))
```

yields

```
["a",0]
["b",0]
["b",0,"a",1]
["b",0,"b",1]
```

in jq. Here, we can see that jq actually yields both outputs of `UPDATE`, namely `["a",0]` and `["b",0]`, but it only uses the last of them, namely `["b",0]`, for the second iteration.

In contrast, jaq yields:

```
["a",0]
["a",0,"a",1]
["a",0,"b",1]
["b",0]
["b",0,"a",1]
["b",0,"b",1]
```

Examples

Filter	`foreach .[] as $item (0; . + $item)`
Input	`[1,2,3,4,5]`

Output	`1`
	`3`
	`6`
	`10`
	`15`

Run	

Filter	`foreach .[] as $item (0; . + $item; [$item, . * 2])`
Input	`[1,2,3,4,5]`

Output	`[1,2]`
	`[2,6]`
	`[3,12]`
	`[4,20]`
	`[5,30]`

Run	

Filter	`foreach .[] as $item (0; . + 1; {index: ., $item})`
Input	`["foo", "bar", "baz"]`
Output	`{"index":1,"item":"foo"}` `{"index":2,"item":"bar"}` `{"index":3,"item":"baz"}`
Run	

Definitions

When you have a filter `g`, you can give it a name `f` as follows:

```
def f: g;
```

This is called a *function definition*. Many builtin functions are implemented by definition.

Example

The definition `def increment: . + 1;` gives the filter `. + 1` the name `increment`.

A function definition `def f: g;` that is followed by a filter `h` is a filter in which both `g` and `h` may call `f`. (Calls of `f` in `g` are recursive calls.)

Example

The filter `def increment: . + 1; 2 | increment` is equivalent to `2 | . + 1`.

Note

In jq, you can write definitions wherever you can write a filter. That allows definitions in places that might be considered rather unorthodox in other programming languages. For example, you can write `1 + def a: 2; def b: 3; a * b`, which is equivalent to `1 + 2 * 3`.

A function may take arguments, for example:

```
def map(f): [.[] | f];
```

Arguments are passed as *filters* (functions with no arguments), *not* as values. The same argument may be referenced multiple times with different inputs; for example, in `map`, the argument `f` is run for each element of the input array. Arguments to a function work more like callbacks than like value arguments. This is important to understand.

Example

Consider the following filter:

```
def foo(f): f|f;
5|foo(.*2)
```

The result will be 20 because `f` is `.*2`, and during the first invocation of `f` `.` will be 5, and the second time it will be 10 (5 * 2), so the result will be 20.

If you want to pass an argument by value, you can prefix its name with `$`.

Example

The definition

```
def addvalue($f): map(. + $f);
```

is equivalent to

```
def addvalue(f): f as $f | map(. + $f);`
```

With either definition, `addvalue(.foo)` adds the current input's `.foo` field to each element of the input.

Multiple definitions using the same function name are allowed. Each re-definition replaces the previous one for the same number of function arguments, but only for references from functions (or main program) subsequent to the re-definition. See also the section on scoping.

Examples

Filter	`def addvalue(f): . + [f]; map(addvalue(.[0]))`
Input	`[[1,2],[10,20]]`
Output	`[[1,2,1], [10,20,10]]`
Run	

Filter	`def addvalue(f): f as $x | map(. + $x); addvalue(.[0])`
Input	`[[1,2],[10,20]]`
Output	`[[1,2,1,2], [10,20,1,2]]`
Run	

Recursion

Any jq function can be recursive. The subsection on recursion functions gives a few examples, such as `recurse`.

Tail calls are optimized whenever the expression to the left of the recursive call outputs its last value. In practice this means that the expression to the left of the recursive call should not produce more than one output for each input.

Example

The builtin function `repeat` can be naively implemented like `repeat_naive` below. It is tail-recursive, however, it binds `f` to a new argument whenever `repeat_naive` is called recursively. This makes `f` more costly to call with every recursion step. For that reason, `repeat` is implemented like below, where `f` is bound only once, and the recursive call does not have to perform any binding.

```
def repeat_naive(f):
    f, repeat_naive(f);

def repeat(f):
  def _repeat:
    f, _repeat;
  _repeat;
```

Example

The builtin function `while` is also implemented recursively. We apply a similar transformation as above for `repeat` to keep the cost of calls to `cond` and `update` constant:

```
def while_naive(cond; update):
    if cond
    then ., (update | while_naive(cond; update))
    else empty
    end;

def while (cond; update):
```

```
def _while:
  if cond
  then ., (update | _while)
  else empty
  end;
_while
```

Generators and iterators

Some jq operators and functions are actually generators in that they can produce zero, one, or more values for each input, just as one might expect in other programming languages that have generators. For example, `.[]` generates all the values in its input (which must be an array or an object), `range(0; 10)` generates the integers between 0 and 10, and so on.

Even the comma operator is a generator, generating first the values generated by the expression to the left of the comma, then the values generated by the expression on the right of the comma.

The `empty` builtin is the generator that produces zero outputs. The `empty` builtin backtracks to the preceding generator expression.

All jq functions can be generators just by using builtin generators. It is also possible to construct new generators using only recursion and the comma operator. If recursive calls are "in tail position" then the generator will be efficient. In the example below the recursive call by `_range` to itself is in tail position. The example shows off three advanced topics: tail recursion, generator construction, and sub-functions.

Examples

Filter	`def range(init; upto; by): def _range: if (by > 0 and . < upto) or (by < 0 and . > upto) then ., ((.+by)|_range) else . end; if by == 0 then init else init|_range end | select((by > 0 and . < upto) or (by < 0 and . > upto)); range(0; 10; 3)`
Input	`null`

Output	`0`
	`3`
	`6`
	`9`

Run	

Filter	`def while(cond; update): def _while: if cond then ., (update | _while) else empty end; _while; [while(.<100; .*2)]`
Input	`1`
Output	`[1,2,4,8,16,32,64]`
Run	

Assignment

jq provides a number of binary assignment operators, such as `|=` and `=`. These replace parts of the input at positions given by the left-hand side with outputs given by the right-hand side, then return the updated input.

Example

The filter `{a: 1, b: 2} | (.a = 3)` outputs `{a: 3, b: 2}`. Here, we replaced the value at position `.a` with 3.

All values in jq are immutable. That means that the input to an assignment is not actually changed; instead, you can think of an assignment creating a *copy* of its input before changing it, then returning the changed copy. The original input remains the same.

Example

> The filter `{a:{b:{c:1}}} | (.a.b = 3), .` outputs `{"a":{"b":3}}` and `{"a":{"b":{"c":1}}}`, because the last sub-expression, `.`, sees the original value, not the modified value.

We can use any kind of compound path that starts with `.` on the left-hand side of an assignment, such as `.[].a` or `.[0]`. We'll discuss usage of other filters on the left-hand side in complex assignments.

Update assignment: `|=`

For every position returned by `p`, the update operator `p |= f` replaces the value `v` at that position by the output of `f` applied to `v`.

Example

> The filter `{foo: 1, bar: 3} | .foo |= .+1` builds an object with the `foo` field set to the input's `foo` plus 1, resulting in the output `{foo: 2, bar: 3}`.

Example

> The filter `[1, 2, 3] | .[] |= . + 1` returns `[2, 3, 4]`. Here, `.[]` returns multiple positions, and the values at each of these positions are updated with `. + 1`.

If the right-hand side outputs no values (i.e., `empty`), then the value at the current position is deleted, as with `del(path)`.

Example

> The filter `{a: 1, b: 2} | .a |= empty` returns `{b: 2}`.

Example

> The filter `[1, 2, 3] | .[0] |= empty` returns `[2, 3]`.

Example

> The filter `[1, 2, 3, 4] | .[] |= select(. % 2 == 0)` returns `[2, 4]`. That means that we can use assignments to filter values from arrays and objects.

If the right-hand side outputs multiple values, only the first output is used.

Example

> The filter `{a: 1} | .a |= (2, 3)` yields `{a: 2}`.

Compatibility

> In jq 1.5 and earlier releases, only the last output was used.

Plain assignment: `=`

The plain assignment operator `=` differs from `|=` in two main points: First, the input to the right-hand side is the same as the input to the left-hand side, not the current value returned by the left-hand side. Second, when the right-hand side returns multiple values, then the operation is performed for each of these values.

Example

To see the difference between = and `|=`, let us provide the input `{"a": {"b": 10}, "b": 20}` to the programs `.a = .b` and `.a |= .b`. The former sets the `a` field of the input to the `b` field of the input, producing the output `{"a": 20, "b": 20}`. The latter sets the `a` field of the input to the `a` field's `b` field, producing `{"a": 10, "b": 20}`.

Example

The filter `{a: 1} | .a = (2, 3)` yields two outputs, namely `{a: 2}` and `{a: 3}`.

Note

The filter `a = b` is equivalent to `b as $x | a |= $x` (where `$x` is a fresh variable name).

Note

Assignment works a little differently in jq than in most programming languages. jq does not distinguish between references to and copies of something — two objects or arrays are either equal or not equal, without any further notion of being "the same object" or "not the same object".

If an object has two fields, `.foo` and `.bar`, and you set `.bar = .foo`, then changing `.foo` does not impact `.bar`. If you're used to programming in languages like Python, Java, Ruby, JavaScript, etc., then you can think of it as though jq does a full deep copy of every object before it does the assignment (for performance it doesn't actually do that, but that's the general idea).

This means that it's impossible to build circular values in jq (such as an array whose first element is itself). This is quite intentional, and ensures that anything a jq program produces can be represented in JSON.

Most users will want to use modification assignment operators, such as `|=` or `+=`, rather than =.

Examples

Filter	`.a = .b`
Input	`{"a": {"b": 10}, "b": 20}`
Output	`{"a":20,"b":20}`
Run	

Filter	`.a |= .b`
Input	`{"a": {"b": 10}, "b": 20}`
Output	`{"a":10,"b":20}`
Run	

Arithmetic update assignment: +=, -=, *=, /=, %=, //=

jq has a few operators of the form `a op= b`. So, `+= 1` can be used to increment values, being the same as `|= . + 1`.

Like =, the right-hand side of an arithmetic update operator receives the same input as the left-hand side, and when the right-hand side returns multiple values, then the operation is performed for each of these values.

Example

The filter `{a: 1, b: 2} | .a += .b` yields `{a: 3, b: 2}`, because `.b` was executed on the original input (`{a: 1, b: 2}`), not on the value that it updated (`1`). In contrast, `{a: 1, b: 2} | .a |= . + .b` yields an error, because `.b` is executed on the value `1` found at the position `.a`.

Example

The filter `{a: 1} | .a += (1, 2)` yields two outputs, namely `{a: 2}` and `{a: 3}`.

Note

For any arithmetic operation `op`, the filter `a op= b` is equivalent to `b as $x | a |= . op $x`.

Examples

Filter	`.foo += 1`
Input	`{"foo": 42}`
Output	`{"foo": 43}`
Run	

Complex assignments

jq accepts far more expressions on the left-hand side of assignments than most languages. So far, we have seen assignments using simple path operators such as `.[0]` and `.a` on the left-hand side. We are now going to show more complex filters on the left-hand side.

First, we can write any compound path on the left-hand side of an update.

Example

Suppose that the input is an object with a field “posts” which is an array of posts. The filter `.posts[0].title = "JQ Manual"` sets the “title” field of the first post.

Example

The filter `.posts[].comments += ["this is great"]` appends the string “this is great” to the “comments” array of *each* post in the input.

In general, on the left-hand side of an assignment, we can use filters that evaluate to a *concatenation of compound paths*, where each of these compound paths must start with `.`. We call such filters *path expressions*.

When jq evaluates an assignment, it tries to evaluate its left-hand side to a concatenation of compound paths. If it succeeds, it updates the values at the positions corresponding to these paths.

Example

Suppose we want to add a comment to blog posts, using the same “blog” input as above. This time, we only want to comment on the posts written by “stedolan”. We can find the comments for these posts using the “select” function described earlier:

```
.posts[] | select(.author == "stedolan") | .comments
```

We can evaluate this to a concatenation of compound paths — for example, if the 3rd and 42th post were written by “stedolan”, this would yield `.posts[3].comments, .posts[42].comments`. We can therefore use this on the left-hand side of an assignment, such as:

```
(.posts[] | select(.author == "stedolan") | .comments) += ["terrible."]
```

Example

The filter `$var.foo = 1` yields an error, because `$var.foo` is a compound path that starts with `$var`, not with `.`. Therefore, this path does not point to the input of the assignment. You can use `$var | .foo = 1` instead.

Example

The filter `{foo: 1, bar: 2} | (.foo, .bar) |= .+1` builds an object with the `foo` field set to the input's `foo` plus 1, and the `bar` field set to the input's `bar` plus 1. Its output is `{foo: 2, bar: 3}`.

Note

Due to precedence rules, `.a,.b=0` does not set `.a` and `.b`, because it is equivalent to `.a, (.b=0)`. The filter `(.a,.b)=0` sets both.

Examples

Filter	`(..|select(type=="boolean")) |= if . then 1 else 0 end`
Input	`[true,false,[5,true,[true,[false]],false]]`
Output	`[1,0,[5,1,[1,[0]],0]]`
Run	

Filter	`(.a, .b) = range(3)`
Input	`{}`

Output	`{"a":0,"b":0}`
	`{"a":1,"b":1}`
	`{"a":2,"b":2}`

Run	

Filter	`(.a, .b) |= range(3)`
Input	`{}`
Output	`{"a":0,"b":0}`
Run	

Path expressions

We now show which kinds of filters are path expressions, i.e. which filters can be used on the left-hand side of assignments.

The following filters are path expressions:

- `.` (identity)
- `..` (recursive descent)
- compound path: if it starts with some `f`, then `f` must be a path expression
  - (`.[]` is a path expression because it starts with `.`, which is a path expression)
  - (`{}[]` is *not* a path expression, because it starts with `{}`, which is no path expression)
- `if i then t else e end`: if `i` and `e` are path expressions
- `f as $x | g`: if `g` is a path expression
- `f, g`: if `f` and `g` are path expressions
- `f | g`: if `f` and `g` are path expressions
- `f // g`: if `f` and `g` are path expressions
- `f?`: if `f` is a path expression

- `label $x | f`: if `f` is a path expression
- `break $x`
- `def f: g; h` (function definition): if `h` is a path expression

On the contrary, the following filters output values which do *not* point to a part of their input, therefore they are *no* path expressions:

- new values, e.g. `1`, “Hello world”, `[1, 2]`, `{a: 1}`
- arithmetic and comparison operations, e.g. `. + 1`
- `and`, `or`
- `$x` (variable)
- assignment (`|=`, `=`, `+=`, …)

For function calls, it depends on the function: If the function is implemented by definition and its definition is a path expression, then the function call is a path expression as well. For example, this is the case for select) and recurse. However, most builtin functions return outputs that do not point to a part of their input, so calls to them are no path expressions.

Note

> This characterisation of path expressions is an *underapproximation*; that is, there are filters that do not correspond to these criteria, yet they can be used on the left-hand side of assignments. For example, our criteria do not say that the filter `if true then empty else 0 end` is a path expression, because `0` is not a path expression. Despite this, we can happily use this filter on the left-hand side of an assignment. Such an assignment will always return its input, because `if true then empty else 0 end` always evaluates to `empty`, so jq does not attempt to evaluate `0` as path.

Compatibility

> jaq’s approach to handling assignments is quite different from that of jq and gojq. Specifically, jaq executes assignments without constructing compound paths. This means that jaq does not allow certain filters on the left-hand side of assignments, notably `f?` and `label $x | f`. jaq’s approach is generally more performant, but in certain scenarios, jaq and jq will produce different results, in particular when using `f |= empty`. However, for the examples in this section, jq and jaq yield the same outputs.

Managing large programs

Comments

You can write comments in your jq programs using `#`.

A `#` character (not part of a string) starts a comment. All characters from `#` to the end of the line are ignored.

If the end of the line is preceded by an odd number of backslash characters, the following line is also considered part of the comment and is ignored.

For example, the following code outputs `[1,3,4,7]`

```
[
  1,
  # foo \
  2,
  # bar \\
  3,
  4, # baz \\\
  5, \
  6,
  7
  # comment \
    comment \
    comment
]
```

Note

A backslash continuing the comment on the next line can be useful when writing the "shebang" for a jq script:

```
#!/bin/sh --
# total - Output the sum of the given arguments (or stdin)
# usage: total [numbers...]
# \
exec jq --args -MRnf "$0" -- "$@"

$ARGS.positional |
reduce (
  if . == []
    then inputs
    else .[]
  end |
  . as $dot |
  try tonumber catch false |
  if not or isnan then
    @json "total: Invalid number \($dot).\n" | halt_error(1)
  end
) as $n (0; . + $n)
```

The `exec` line is considered a comment by jq, so it is ignored. But it is not ignored by `sh`, since in `sh` a backslash at the end of the line does not continue the comment. With this trick, when the script is invoked as `total 1 2`, `/bin/sh -- /path/to/total 1 2` will be run, and `sh` will then run `exec jq --args -MRnf /path/to/total -- 1 2` replacing itself with a `jq` interpreter invoked with the specified options (`-M`, `-R`, `-n`, `--args`), that evaluates the current file (`$0`), with the arguments (`$@`) that were passed to `sh`.

Compatibility

jaq ignores backslashes at the end of comment lines.

Modules

jq has a library/module system. Modules are files whose names end in `.jq`.

Importing / including modules

The directives

```
import RelativePathString as NAME [<metadata>];
include RelativePathString [<metadata>];
```

import a module found at the given path relative to a directory in a search path. A `.jq` suffix will be added to the relative path string. If `import` is used, the module's symbols are prefixed with `NAME::`. If `include` is used, the module's symbols are imported into the caller's namespace.

The optional metadata must be a constant jq expression. It should be an object with keys like `homepage` and so on. At this time jq only uses the `search` key/value of the metadata. The metadata is also made available to users via the `modulemeta` builtin.

The `search` key in the metadata, if present, should have a string or array value (array of strings); this is the search path to be prefixed to the top-level search path.

Importing data

The directive

```
import RelativePathString as $NAME [<metadata>];
```

imports a JSON file found at the given path relative to a directory in a search path. A `.json` suffix will be added to the relative path string. The file's data will be available as `$NAME::NAME`.

The optional metadata is considered the same way as module imports.

Providing module metadata

The directive

```
module <metadata>;
```

may be put at the beginning of a module file. It is entirely optional and serves only the purpose of providing metadata that can be read with the `modulemeta` builtin.

The metadata must be a constant jq expression. It should be an object with keys like `homepage`. At this time jq doesn't use this metadata, but it is made available to users via the `modulemeta` builtin.

Search paths

Modules imported by a program are searched for in a default search path (see below). The `import` and `include` directives allow the importer to alter this path.

Paths in the search path are subject to various substitutions:

- For paths starting with `~/`, the user's home directory is substituted for `~`.
- For paths starting with `$ORIGIN/`, the directory where the jq executable is located is substituted for `$ORIGIN`.
- For paths starting with `./` or paths that are `.`, the path of the including file is substituted for `.`. For top-level programs given on the command-line, the current directory is used.

Import directives can optionally specify a search path to which the default is appended.

The default search path is the search path given to the `-L` command-line option, else `["~/.jq", "$ORIGIN/../lib/jq", "$ORIGIN/../lib"]`.

Null and empty string path elements terminate search path processing.

A dependency with relative path `foo/bar` would be searched for in `foo/bar.jq` and `foo/bar/bar.jq` in the given search path. This is intended to allow modules to be placed in a directory along with, for example, version control files, README files, and so on, but also to allow for single-file modules.

Consecutive components with the same name are not allowed to avoid ambiguities (e.g., `foo/foo`).

For example, with `-L$HOME/.jq` a module `foo` can be found in `$HOME/.jq/foo.jq` and `$HOME/.jq/foo/foo.jq`.

If `.jq` exists in the user's home directory, and is a file (not a directory), it is automatically sourced into the main program.

Builtin functions

This section documents all named filter functions that are available by default in any jq program.

Basic functions

`empty`

`empty` returns no results. None at all. Not even `null`.

It's useful on occasion. You'll know if you need it :)

Examples

Filter	`1, empty, 2`
Input	`null`

Output	`1`
	`2`

Run	

Filter	`[1,2,empty,3]`
Input	`null`
Output	`[1,2,3]`
Run	

`error`, `error(message)`

Produces an error with the input value, or with the message given as the argument. Errors can be caught with try/catch.

Examples

Filter	`try error catch .`
Input	`"error message"`
Output	`"error message"`
Run	

Filter	`try error("invalid value: \(.)") catch .`
Input	`42`
Output	`"invalid value: 42"`
Run	

`length`

The `length` function gets the length of various different types of values:

- The length of a **string** is the number of Unicode codepoints it contains (which will be the same as its JSON-encoded length in bytes if it's pure ASCII).
- The length of a **number** is its absolute value.
- The length of an **array** is the number of elements.
- The length of an **object** is the number of key-value pairs.
- The length of **null** is zero.
- It is an error to use `length` on a **boolean**.

Examples

Filter	`.[] | length`
Input	`[[1,2], "string", {"a":2}, null, -5]`

Output	`2`
	`6`
	`1`
	`0`
	`5`

Run	

`keys`, `keys_unsorted`

The builtin function `keys`, when given an object, returns its keys in an array.

The keys are sorted “alphabetically”, by unicode codepoint order. This is not an order that makes particular sense in any particular language, but you can count on it being the same for any two objects with the same set of keys, regardless of locale settings.

When `keys` is given an array, it returns the valid indices for that array: the integers from 0 to length-1.

The `keys_unsorted` function is just like `keys`, but if the input is an object then the keys will not be sorted, instead the keys will roughly be in insertion order.

Examples

Filter	`keys`
Input	`{"abc": 1, "abcd": 2, "Foo": 3}`
Output	`["Foo", "abc", "abcd"]`
Run	

Filter	`keys`
Input	`[42,3,35]`
Output	`[0,1,2]`
Run	

`map(f)`, `map_values(f)`

For any filter `f`, `map(f)` and `map_values(f)` apply `f` to each of the values in the input array or object, that is, to the values of `.[]`.

In the absence of errors, `map(f)` always outputs an array whereas `map_values(f)` outputs an array if given an array, or an object if given an object.

When the input to `map_values(f)` is an object, the output object has the same keys as the input object except for those keys whose values when piped to `f` produce no values at all.

The key difference between `map(f)` and `map_values(f)` is that the former simply forms an array from all the values of `($x|f)` for each value, `$x`, in the input array or object, but `map_values(f)` only uses `first($x|f)`.

Specifically, for object inputs, `map_values(f)` constructs the output object by examining in turn the value of `first(.[$k]|f)` for each key, `$k`, of the input. If this expression produces no values, then the corresponding key will be dropped; otherwise, the output object will have that value at the key, `$k`.

Here are some examples to clarify the behavior of `map` and `map_values` when applied to arrays. These examples assume the input is `[1]` in all cases:

```
map(.+1)          #=>  [2]
map(., .)         #=>  [1,1]
map(empty)        #=>  []

map_values(.+1)   #=>  [2]
map_values(., .)  #=>  [1]
map_values(empty) #=>  []
```

`map(f)` is equivalent to `[.[] | f]` and `map_values(f)` is equivalent to `.[] |= f`.

In fact, these are their implementations.

Examples

Filter	`map(.+1)`
Input	`[1,2,3]`
Output	`[2,3,4]`
Run	

Filter	`map_values(.+1)`
Input	`{"a": 1, "b": 2, "c": 3}`
Output	`{"a": 2, "b": 3, "c": 4}`
Run	

Filter	`map(., .)`
Input	`[1,2]`
Output	`[1,1,2,2]`
Run	

Filter	`map_values(. // empty)`
Input	`{"a": null, "b": true, "c": false}`
Output	`{"b":true}`
Run	

`to_entries`, `from_entries`, `with_entries(f)`

These functions convert between an object and an array of key-value pairs. If `to_entries` is passed an object, then for each `k: v` entry in the input, the output array includes `{"key": k, "value": v}`.

`from_entries` does the opposite conversion, and `with_entries(f)` is a shorthand for `to_entries | map(f) | from_entries`, useful for doing some operation to all keys and values of an object. `from_entries` accepts `"key"`, `"Key"`, `"name"`, `"Name"`, `"value"`, and `"Value"` as keys.

Examples

Filter	`to_entries`
Input	`{"a": 1, "b": 2}`
Output	`[{"key":"a", "value":1}, {"key":"b", "value":2}]`
Run	

Filter	`from_entries`
Input	`[{"key":"a", "value":1}, {"key":"b", "value":2}]`
Output	`{"a": 1, "b": 2}`
Run	

Filter	`with_entries(.key |= "KEY_" + .)`
Input	`{"a": 1, "b": 2}`
Output	`{"KEY_a": 1, "KEY_b": 2}`
Run	

`not`

The function `not` negates the boolean value of its input. It is defined as:

```
def not: if . then false else true;
```

`select(boolean_expression)`

The function `select(f)` produces its input unchanged if `f` returns true for that input, and produces no output otherwise.

It's useful for filtering lists: `[1,2,3] | map(select(. >= 2))` will give you `[2,3]`.

Examples

Filter	`map(select(. >= 2))`
Input	`[1,5,3,0,7]`
Output	`[5,3,7]`
Run	

Filter	`.[] | select(.id == "second")`
Input	`[{"id": "first", "val": 1}, {"id": "second", "val": 2}]`
Output	`{"id": "second", "val": 2}`
Run	

`type`

The `type` function returns the type of its argument as a string, which is one of null, boolean, number, string, array or object.

Examples

Filter	`map(type)`
Input	`[0, false, [], {}, null, "hello"]`
Output	`["number", "boolean", "array", "object", "null", "string"]`
Run	

`arrays`, **`objects`**, **`iterables`**, **`booleans`**, **`numbers`**, **`normals`**, **`finites`**, **`strings`**, **`nulls`**, **`values`**, **`scalars`**

These built-ins select only inputs that are arrays, objects, iterables (arrays or objects), booleans, numbers, normal numbers, finite numbers, strings, null, non-null values, and non-iterables, respectively.

Examples

Filter	`.[]|numbers`
Input	`[[],{},1,"foo",null,true,false]`
Output	`1`
Run	

Membership functions

`contains(element)`

The filter `contains(b)` will produce true if b is completely contained within the input. A string B is contained in a string A if B is a substring of A. An array B is contained in an array A if all elements in B are contained in any element in A. An object B is contained in object A if all of the values in B are contained in the value in A with the same key. All other types are assumed to be contained in each other if they are equal.

Examples

Filter	`contains("bar")`
Input	`"foobar"`
Output	`true`
Run	

Filter	`contains(["baz", "bar"])`
Input	`["foobar", "foobaz", "blarp"]`
Output	`true`
Run	

Filter	`contains(["bazzzzz", "bar"])`
Input	`["foobar", "foobaz", "blarp"]`
Output	`false`
Run	

Filter	`contains({foo: 12, bar: [{barp: 12}]})`
Input	`{"foo": 12, "bar":[1,2,{"barp":12, "blip":13}]}`
Output	`true`
Run	

Filter	`contains({foo: 12, bar: [{barp: 15}]})`
Input	`{"foo": 12, "bar":[1,2,{"barp":12, "blip":13}]}`
Output	`false`
Run	

`indices(s)`

Outputs an array containing the indices in `.` where `s` occurs. The input may be an array, in which case if `s` is an array then the indices output will be those where all elements in `.` match those of `s`.

Examples

Filter	`indices(", ")`
Input	`"a,b, cd, efg, hijk"`
Output	`[3,7,12]`
Run	

Filter	`indices(1)`
Input	`[0,1,2,1,3,1,4]`
Output	`[1,3,5]`
Run	

Filter	`indices([1,2])`
Input	`[0,1,2,3,1,4,2,5,1,2,6,7]`
Output	`[1,8]`
Run	

`index(s)`, `rindex(s)`

Outputs the index of the first (`index`) or last (`rindex`) occurrence of `s` in the input.

Examples

Filter	`index(", ")`
Input	`"a,b, cd, efg, hijk"`
Output	`3`
Run	

Filter	`index(1)`
Input	`[0,1,2,1,3,1,4]`
Output	`1`
Run	

Filter	`index([1,2])`
Input	`[0,1,2,3,1,4,2,5,1,2,6,7]`
Output	`1`
Run	

Filter	`rindex(", ")`
Input	`"a,b, cd, efg, hijk"`
Output	`12`
Run	

Filter	`rindex(1)`
Input	`[0,1,2,1,3,1,4]`
Output	`5`
Run	

Filter	`rindex([1,2])`
Input	`[0,1,2,3,1,4,2,5,1,2,6,7]`
Output	`8`
Run	

`inside`

The filter `inside(b)` will produce true if the input is completely contained within b. It is, essentially, an inversed version of `contains`.

Examples

Filter	`inside("foobar")`
Input	`"bar"`
Output	`true`
Run	

Filter	`inside(["foobar", "foobaz", "blarp"])`
Input	`["baz", "bar"]`
Output	`true`
Run	

Filter	`inside(["foobar", "foobaz", "blarp"])`
Input	`["bazzzzz", "bar"]`
Output	`false`
Run	

Filter	`inside({"foo": 12, "bar":[1,2,{"barp":12, "blip":13}]})`
Input	`{"foo": 12, "bar": [{"barp": 12}]}`
Output	`true`
Run	

Filter	`inside({"foo": 12, "bar":[1,2,{"barp":12, "blip":13}]})`
Input	`{"foo": 12, "bar": [{"barp": 15}]}`
Output	`false`
Run	

`has(key)`

The builtin function `has` returns whether the input object has the given key, or the input array has an element at the given index.

`has($key)` has the same effect as checking whether `$key` is a member of the array returned by `keys`, although `has` will be faster.

Examples

Filter	`map(has("foo"))`
Input	`[{"foo": 42}, {}]`
Output	`[true, false]`
Run	

Filter	`map(has(2))`
Input	`[[0,1], ["a","b","c"]]`
Output	`[false, true]`
Run	

`in`

The builtin function `in` returns whether or not the input key is in the given object, or the input index corresponds to an element in the given array. It is, essentially, an inversed version of `has`.

Examples

Filter	`.[] | in({"foo": 42})`
Input	`["foo", "bar"]`
Output	`true` `false`
Run	

Filter	`map(in([0,1]))`
Input	`[2, 0]`
Output	`[false, true]`
Run	

`bsearch(x)`

`bsearch(x)` conducts a binary search for x in the input array. If the input is sorted and contains x, then `bsearch(x)` will return its index in the array; otherwise, if the array is sorted, it will return (-1 - ix) where ix is an insertion point such that the array would still be sorted after the insertion of x at ix. If the array is not sorted, `bsearch(x)` will return an integer that is probably of no interest.

Examples

Filter	`bsearch(0)`
Input	`[0,1]`
Output	`0`
Run	

Filter	`bsearch(0)`
Input	`[1,2,3]`
Output	`-1`
Run	

Filter	`bsearch(4) as $ix | if $ix < 0 then .[-(1+$ix)] = 4 else . end`
Input	`[1,2,3]`
Output	`[1,2,3,4]`
Run	

Path expression functions

The following functions all take a path expression.

`path(path_expression)`

Outputs array representations of the given path expression in `.`. The outputs are arrays of strings (object keys) and/or numbers (array indices). The outputs of this function can be processed with path functions.

Path expressions are jq expressions like `.a`, but also `.[]`. There are two types of path expressions: ones that can match exactly, and ones that cannot. For example, `.a.b.c` is an exact match path expression, while `.a[].b` is not.

`path(exact_path_expression)` will produce the array representation of the path expression even if it does not exist in `.`, if `.` is `null` or an array or an object.

`path(pattern)` will produce array representations of the paths matching `pattern` if the paths exist in `.`.

Note that the path expressions are not different from normal expressions. The expression `path(..|select(type=="boolean"))` outputs all the paths to boolean values in `.`, and only those paths.

Examples

Filter	`path(.a[0].b)`
Input	`null`
Output	`["a",0,"b"]`
Run	

Filter	`[path(..)]`
Input	`{"a":[{"b":1}]}`
Output	`[[],["a"],["a",0],["a",0,"b"]]`
Run	

`del(path_expression)`

The builtin function `del` removes a key and its corresponding value from an object.

Examples

Filter	`del(.foo)`
Input	`{"foo": 42, "bar": 9001, "baz": 42}`
Output	`{"bar": 9001, "baz": 42}`
Run	

Filter	`del(.[1, 2])`
Input	`["foo", "bar", "baz"]`
Output	`["foo"]`
Run	

`pick(pathexps)`

Emit the projection of the input object or array defined by the specified sequence of path expressions, such that if `p` is any one of these specifications, then `(. | p)` will evaluate to the same value as `(. | pick(pathexps) | p)`. For arrays, negative indices and `.[m:n]` specifications should not be used.

Examples

Filter	`pick(.a, .b.c, .x)`
Input	`{"a": 1, "b": {"c": 2, "d": 3}, "e": 4}`
Output	`{"a":1,"b":{"c":2},"x":null}`
Run	

Filter	`pick(.[2], .[0], .[0])`
Input	`[1,2,3,4]`
Output	`[1,null,3]`
Run	

Path functions

The following functions all produce or process paths in the format output by the `path` function.

`paths`, `paths(node_filter)`

`paths` outputs the paths to all the elements in its input (except it does not output the empty list, representing . itself).

`paths(f)` outputs the paths to any values for which `f` is `true`. That is, `paths(type == "number")` outputs the paths to all numeric values.

Examples

Filter	`[paths]`
Input	`[1,[[],{"a":2}]]`
Output	`[[0],[1],[1,0],[1,1],[1,1,"a"]]`
Run	

Filter	`[paths(type == "number")]`
Input	`[1,[[],{"a":2}]]`
Output	`[[0],[1,1,"a"]]`
Run	

`getpath(PATHS)`

The builtin function `getpath` outputs the values in `.` found at each path in `PATHS`.

Examples

Filter	`getpath(["a","b"])`
Input	`null`
Output	`null`
Run	

Filter	`[getpath(["a","b"], ["a","c"])]`
Input	`{"a":{"b":0, "c":1}}`
Output	`[0, 1]`
Run	

`setpath(PATHS; VALUE)`

The builtin function `setpath` sets the `PATHS` in `.` to `VALUE`.

Examples

Filter	`setpath(["a","b"]; 1)`
Input	`null`
Output	`{"a": {"b": 1}}`
Run	

Filter	`setpath(["a","b"]; 1)`
Input	`{"a":{"b":0}}`
Output	`{"a": {"b": 1}}`
Run	

Filter	`setpath([0,"a"]; 1)`
Input	`null`
Output	`[{"a":1}]`
Run	

`delpaths(PATHS)`

The builtin function `delpaths` deletes the `PATHS` in `.`. `PATHS` must be an array of paths, where each path is an array of strings and numbers.

Examples

Filter	`delpaths([["a","b"]])`
Input	`{"a":{"b":1},"x":{"y":2}}`
Output	`{"a":{},"x":{"y":2}}`
Run	

Reduction functions

`add`, `add(generator)`

The filter `add` takes as input an array, and produces as output the elements of the array added together. This might mean summed, concatenated or merged depending on the types of the elements of the input array - the rules are the same as those for the + operator (described above).

If the input is an empty array, `add` returns `null`.

`add(generator)` operates on the given generator rather than the input.

Examples

Filter	`add`
Input	`["a","b","c"]`
Output	`"abc"`
Run	

Filter	`add`
Input	`[1, 2, 3]`
Output	`6`
Run	

Filter	`add`
Input	`[]`
Output	`null`
Run	

Filter	`add(.[].a)`
Input	`[{"a":3}, {"a":5}, {"b":6}]`
Output	`8`
Run	

`any`, `any(condition)`, `any(generator; condition)`

The filter `any` takes as input an array of boolean values, and produces `true` as output if any of the elements of the array are `true`.

If the input is an empty array, `any` returns `false`.

The `any(condition)` form applies the given condition to the elements of the input array.

The `any(generator; condition)` form applies the given condition to all the outputs of the given generator.

Examples

Filter	`any`
Input	`[true, false]`
Output	`true`
Run	

Filter	`any`
Input	`[false, false]`
Output	`false`
Run	

Filter	`any`
Input	`[]`
Output	`false`
Run	

`all`, `all(condition)`, `all(generator; condition)`

The filter `all` takes as input an array of boolean values, and produces `true` as output if all of the elements of the array are `true`.

The `all(condition)` form applies the given condition to the elements of the input array.

The `all(generator; condition)` form applies the given condition to all the outputs of the given generator.

If the input is an empty array, `all` returns `true`.

Examples

Filter	`all`
Input	`[true, false]`
Output	`false`
Run	

Filter	`all`
Input	`[true, true]`
Output	`true`
Run	

Filter	`all`
Input	`[]`
Output	`true`
Run	

Number functions

jq currently only has IEEE754 double-precision (64-bit) floating point number support.

Besides simple arithmetic operators such as +, jq also has most standard math functions from the C math library. C math functions that take a single input argument (e.g., `sin()`) are available as zero-argument jq functions. C math functions that take two input arguments (e.g., `pow()`) are available as two-argument jq functions that ignore `.`. C math functions that take three input arguments are available as three-argument jq functions that ignore `.`.

Availability of standard math functions depends on the availability of the corresponding math functions in your operating system and C math library. Unavailable math functions will be defined but will raise an error.

One-input C math functions: `acos` `acosh` `asin` `asinh` `atan` `atanh` `cbrt` `ceil` `cos` `cosh` `erf` `erfc` `exp` `exp10` `exp2` `expm1` `fabs` `floor` `gamma` `j0` `j1` `lgamma` `log` `log10` `log1p` `log2` `logb` `nearbyint` `rint` `round` `significand` `sin` `sinh` `sqrt` `tan` `tanh` `tgamma` `trunc` `y0` `y1`.

Two-input C math functions: `atan2` `copysign` `drem` `fdim` `fmax` `fmin` `fmod` `frexp` `hypot` `jn` `ldexp` `modf` `nextafter` `nexttoward` `pow` `remainder` `scalb` `scalbln` `yn`.

Three-input C math functions: `fma`.

See your system's manual for more information on each of these.

`abs`

The builtin function `abs` is defined naively as: `if . < 0 then - . else . end`.

For numeric input, this is the absolute value. See the section on the identity filter for the implications of this definition for numeric input.

To compute the absolute value of a number as a floating point number, you may wish use `fabs`.

Examples

Filter	`map(abs)`
Input	`[-10, -1.1, -1e-1]`
Output	`[10,1.1,1e-1]`
Run	

`floor`

The `floor` function returns the floor of its numeric input.

Examples

Filter	`floor`
Input	`3.14159`
Output	`3`
Run	

`sqrt`

The `sqrt` function returns the square root of its numeric input.

Examples

Filter	`sqrt`
Input	`9`
Output	`3`
Run	

`infinite`, `nan`, `isinfinite`, `isnan`, `isfinite`, `isnormal`

Some arithmetic operations can yield infinities and "not a number" (NaN) values. The `isinfinite` builtin returns `true` if its input is infinite. The `isnan` builtin returns `true` if its input is a NaN. The `infinite` builtin returns a positive infinite value. The `nan` builtin returns a NaN. The `isnormal` builtin returns true if its input is a normal number.

Note that division by zero raises an error.

Currently most arithmetic operations operating on infinities, NaNs, and sub-normals do not raise errors.

Examples

Filter	`.[] | (infinite * .) < 0`
Input	`[-1, 1]`
Output	`true` `false`
Run	

Filter	`infinite, nan | type`
Input	`null`
Output	`"number"` `"number"`
Run	

Array functions

`sort`, `sort_by(path_expression)`

The `sort` functions sorts its input, which must be an array. Values are sorted using the order given by <.

`sort_by` may be used to sort by a particular field of an object, or by applying any jq filter. `sort_by(f)` compares two elements by comparing the result of `f` on each element. When `f` produces multiple values, it firstly compares the first values, and the second values if the first values are equal, and so on.

Examples

Filter	`sort`
Input	`[8,3,null,6]`
Output	`[null,3,6,8]`
Run	

Filter	`sort_by(.foo)`
Input	`[{"foo":4, "bar":10}, {"foo":3, "bar":10}, {"foo":2, "bar":1}]`
Output	`[{"foo":2, "bar":1}, {"foo":3, "bar":10}, {"foo":4, "bar":10}]`
Run	

Filter	`sort_by(.foo, .bar)`
Input	`[{"foo":4, "bar":10}, {"foo":3, "bar":20}, {"foo":2, "bar":1}, {"foo":3, "bar":10}]`
Output	`[{"foo":2, "bar":1}, {"foo":3, "bar":10}, {"foo":3, "bar":20}, {"foo":4, "bar":10}]`
Run	

`group_by(path_expression)`

`group_by(.foo)` takes as input an array, groups the elements having the same `.foo` field into separate arrays, and produces all of these arrays as elements of a larger array, sorted by the value of the `.foo` field.

Any jq expression, not just a field access, may be used in place of `.foo`. The sorting order is the same as described in the `sort` function above.

Examples

Filter	`group_by(.foo)`
Input	`[{"foo":1, "bar":10}, {"foo":3, "bar":100}, {"foo":1, "bar":1}]`
Output	`[[{"foo":1, "bar":10}, {"foo":1, "bar":1}], [{"foo":3, "bar":100}]]`
Run	

`min`, `max`, `min_by(path_exp)`, `max_by(path_exp)`

Find the minimum or maximum element of the input array.

The `min_by(path_exp)` and `max_by(path_exp)` functions allow you to specify a particular field or property to examine, e.g. `min_by(.foo)` finds the object with the smallest `foo` field.

Examples

Filter	`min`
Input	`[5,4,2,7]`
Output	`2`
Run	

Filter	`max_by(.foo)`
Input	`[{"foo":1, "bar":14}, {"foo":2, "bar":3}]`
Output	`{"foo":2, "bar":3}`
Run	

`unique`, `unique_by(path_exp)`

The `unique` function takes as input an array and produces an array of the same elements, in sorted order, with duplicates removed.

The `unique_by(path_exp)` function will keep only one element for each value obtained by applying the argument. Think of it as making an array by taking one element out of every group produced by `group`.

Examples

Filter	`unique`
Input	`[1,2,5,3,5,3,1,3]`
Output	`[1,2,3,5]`
Run	

Filter	`unique_by(.foo)`
Input	`[{"foo": 1, "bar": 2}, {"foo": 1, "bar": 3}, {"foo": 4, "bar": 5}]`
Output	`[{"foo": 1, "bar": 2}, {"foo": 4, "bar": 5}]`
Run	

Filter	`unique_by(length)`
Input	`["chunky", "bacon", "kitten", "cicada", "asparagus"]`
Output	`["bacon", "chunky", "asparagus"]`
Run	

`reverse`

This function reverses an array.

Examples

Filter	`reverse`
Input	`[1,2,3,4]`
Output	`[4,3,2,1]`
Run	

`combinations`, `combinations(n)`

Outputs all combinations of the elements of the arrays in the input array. If given an argument `n`, it outputs all combinations of `n` repetitions of the input array.

Examples

Filter	`combinations`
Input	`[[1,2], [3, 4]]`

Output	`[1, 3]`
	`[1, 4]`
	`[2, 3]`
	`[2, 4]`

Run	

Filter	`combinations(2)`
Input	`[0, 1]`

Output	`[0, 0]`
	`[0, 1]`
	`[1, 0]`
	`[1, 1]`

Run	

`transpose`

Transpose a possibly jagged matrix (an array of arrays). Rows are padded with nulls so the result is always rectangular.

Examples

Filter	`transpose`
Input	`[[1], [2,3]]`
Output	`[[1,2],[null,3]]`
Run	

`flatten`, `flatten(depth)`

The filter `flatten` takes as input an array of nested arrays, and produces a flat array in which all arrays inside the original array have been recursively replaced by their values. You can pass an argument to it to specify how many levels of nesting to flatten.

`flatten(2)` is like `flatten`, but going only up to two levels deep.

Examples

Filter	`flatten`
Input	`[1, [2], [[3]]]`
Output	`[1, 2, 3]`
Run	

Filter	`flatten(1)`
Input	`[1, [2], [[3]]]`
Output	`[1, 2, [3]]`
Run	

Filter	`flatten`
Input	`[[]]`
Output	`[]`
Run	

Filter	`flatten`
Input	`[{"foo": "bar"}, [{"foo": "baz"}]]`
Output	`[{"foo": "bar"}, {"foo": "baz"}]`
Run	

String functions

`utf8bytelength`

The builtin function `utf8bytelength` outputs the number of bytes used to encode a string in UTF-8.

Examples

Filter	`utf8bytelength`
Input	`"\u03bc"`
Output	`2`
Run	

`startswith(str)`

Outputs `true` if . starts with the given string argument.

Examples

Filter	`[.[]|startswith("foo")]`
Input	`["fo", "foo", "barfoo", "foobar", "barfoob"]`
Output	`[false, true, false, true, false]`
Run	

`endswith(str)`

Outputs `true` if . ends with the given string argument.

Examples

Filter	`[.[]|endswith("foo")]`
Input	`["foobar", "barfoo"]`
Output	`[false, true]`
Run	

`ltrimstr(str)`

Outputs its input with the given prefix string removed, if it starts with it.

Examples

Filter	`[.[]|ltrimstr("foo")]`
Input	`["fo", "foo", "barfoo", "foobar", "afoo"]`
Output	`["fo","","barfoo","bar","afoo"]`
Run	

`rtrimstr(str)`
Outputs its input with the given suffix string removed, if it ends with it.

Examples

Filter	`[.[]|rtrimstr("foo")]`
Input	`["fo", "foo", "barfoo", "foobar", "foob"]`
Output	`["fo","","bar","foobar","foob"]`
Run	

`trim`, `ltrim`, `rtrim`
`trim` trims both leading and trailing whitespace.

`ltrim` trims only leading (left side) whitespace.

`rtrim` trims only trailing (right side) whitespace.

Whitespace characters are the usual `" "`, `"\n"` `"\t"`, `"\r"` and also all characters in the Unicode character database with the whitespace property. Note that what considers whitespace might change in the future.

Examples

Filter	`trim, ltrim, rtrim`
Input	`" abc "`

Output	`"abc"`
	`"abc "`
	`" abc"`

Run	

`explode`
Converts an input string into an array of the string’s codepoint numbers.

Examples

Filter	`explode`
Input	`"foobar"`
Output	`[102,111,111,98,97,114]`
Run	

`implode`
The inverse of explode.

Examples

Filter	`implode`
Input	`[65, 66, 67]`
Output	`"ABC"`
Run	

`split(str)`
Splits an input string on the separator argument.

`split` can also split on regex matches when called with two arguments (see the regular expressions section below).

Examples

Filter	`split(", ")`
Input	`"a, b,c,d, e, "`
Output	`["a","b,c,d","e",""]`
Run	

`join(str)`
Joins the array of elements given as input, using the argument as separator. It is the inverse of `split`: that is, running `split("foo") | join("foo")` over any input string returns said input string.

Numbers and booleans in the input are converted to strings. Null values are treated as empty strings. Arrays and objects in the input are not supported.

Compatibility

> When given an array `[x0, x1, ..., xn]`, in jq, `join(x)` converts all elements of the input array to strings and intersperses them with `x`, whereas in jaq, `join(x)` simply calculates `x0 + x + x1 + x + ... + xn`. When all elements of the input array and `x` are strings, jq and jaq yield the same output.

Examples

Filter	`join(", ")`
Input	`["a","b,c,d","e"]`
Output	`"a, b,c,d, e"`
Run	

Filter	`join(" ")`
Input	`["a",1,2.3,true,null,false]`
Output	`"a 1 2.3 true  false"`
Run	

`ascii_downcase`, `ascii_upcase`
Emit a copy of the input string with its alphabetic characters (a-z and A-Z) converted to the specified case.

Examples

Filter	`ascii_upcase`
Input	`"useful but not for é"`
Output	`"USEFUL BUT NOT FOR é"`
Run	

String formatting functions

`@text`

Calls `tostring`, see that function for details.

`@json`

Serializes the input as JSON.

`@html`

Applies HTML/XML escaping, by mapping the characters `<>&'"` to their entity equivalents `&lt;`, `&gt;`, `&amp;`, `&apos;`, `&quot;`.

`@uri`

Applies percent-encoding, by mapping all reserved URI characters to a `%XX` sequence.

`@urid`

The inverse of `@uri`, applies percent-decoding, by mapping all `%XX` sequences to their corresponding URI characters.

`@csv`

The input must be an array, and it is rendered as CSV with double quotes for strings, and quotes escaped by repetition.

`@tsv`

The input must be an array, and it is rendered as TSV (tab-separated values). Each input array will be printed as a single line. Fields are separated by a single tab (ascii `0x09`). Input characters line-feed (ascii `0x0a`), carriage-return (ascii `0x0d`), tab (ascii `0x09`) and backslash (ascii `0x5c`) will be output as escape sequences `\n`, `\r`, `\t`, `\\` respectively.

`@sh`

The input is escaped suitable for use in a command-line for a POSIX shell. If the input is an array, the output will be a series of space-separated strings.

`@base64`

The input is converted to base64 as specified by RFC 4648.

`@base64d`

The inverse of `@base64`, input is decoded as specified by RFC 4648.

Note

> If the decoded string is not UTF-8, the results are undefined.

Recursion functions

`repeat(f)`

The function `repeat(f)` repeatedly runs `f` on the original input. It could be naively defined via:

```
def repeat(f): f, repeat(f)
```

Note

> `repeat(f)` is internally defined as a recursive jq function. Recursive calls within `repeat` will not consume additional memory if `f` produces at most one output for each input. See the section on recursion.

Examples

Filter	`[repeat(.*2, error)?]`
Input	`1`
Output	`[2]`
Run	

`range(upto)`, `range(from; upto)`, `range(from; upto; by)`

The `range` function produces a range of numbers. `range(4; 10)` produces 6 numbers, from 4 (inclusive) to 10 (exclusive). The numbers are produced as separate outputs. Use `[range(4; 10)]` to get a range as an array.

The one argument form generates numbers from 0 to the given number, with an increment of 1.

The two argument form generates numbers from `from` to `upto` with an increment of 1.

The three argument form generates numbers `from` to `upto` with an increment of `by`.

Examples

Filter	`range(2; 4)`
Input	`null`

Output	`2`
	`3`

Run	

Filter	`[range(2; 4)]`
Input	`null`
Output	`[2,3]`
Run	

Filter	`[range(4)]`
Input	`null`
Output	`[0,1,2,3]`
Run	

Filter	`[range(0; 10; 3)]`
Input	`null`
Output	`[0,3,6,9]`
Run	

Filter	`[range(0; 10; -1)]`
Input	`null`
Output	`[]`
Run	

Filter	`[range(0; -5; -1)]`
Input	`null`
Output	`[0,-1,-2,-3,-4]`
Run	

`while(cond; update)`

The `while(cond; update)` function allows you to repeatedly apply an update to `.` until `cond` is false.

Note that `while(cond; update)` is internally defined as a recursive jq function. Recursive calls within `while` will not consume additional memory if `update` produces at most one output for each input. See advanced topics below.

Examples

Filter	`[while(.<100; .*2)]`
Input	`1`
Output	`[1,2,4,8,16,32,64]`
Run	

`until(cond; next)`

The `until(cond; next)` function allows you to repeatedly apply the expression `next`, initially to `.` then to its own output, until `cond` is true. For example, this can be used to implement a factorial function (see below).

Note that `until(cond; next)` is internally defined as a recursive jq function. Recursive calls within `until()` will not consume additional memory if `next` produces at most one output for each input. See advanced topics below.

Examples

Filter	`[.,1]|until(.[0] < 1; [.[0] - 1, .[1] * .[0]])|.[1]`
Input	`4`
Output	`24`
Run	

`recurse(f)`, `recurse`, `recurse(f; condition)`

The `recurse(f)` function allows you to search through a recursive structure, and extract interesting data from all levels. Suppose your input represents a filesystem:

```
{"name": "/", "children": [
  {"name": "/bin", "children": [
    {"name": "/bin/ls", "children": []},
    {"name": "/bin/sh", "children": []}]},
  {"name": "/home", "children": [
    {"name": "/home/stephen", "children": [
      {"name": "/home/stephen/jq", "children": []}]}]}]}
```

Now suppose you want to extract all of the filenames present. You need to retrieve `.name`, `.children[].name`, `.children[].children[].name`, and so on. You can do this with:

```
recurse(.children[]) | .name
```

When called without an argument, `recurse` is equivalent to `recurse(.[]?)`.

`recurse(f)` is identical to `recurse(f; true)` and can be used without concerns about recursion depth.

`recurse(f; condition)` is a generator which begins by emitting . and then emits in turn .|f, .|f|f, .|f|f|f, … so long as the computed value satisfies the condition. For example, to generate all the integers, at least in principle, one could write `recurse(.+1; true)`.

The recursive calls in `recurse` will not consume additional memory whenever `f` produces at most a single output for each input.

Examples

Filter	`recurse(.foo[])`
Input	`{"foo":[{"foo": []}, {"foo":[{"foo":[]}]}]}`

Output	`{"foo":[{"foo":[]},{"foo":[{"foo":[]}]}]}`
	`{"foo":[]}`
	`{"foo":[{"foo":[]}]}`
	`{"foo":[]}`

Run	

Filter	`recurse`
Input	`{"a":0,"b":[1]}`

Output	`{"a":0,"b":[1]}`
	`0`
	`[1]`
	`1`

Run	

Filter	`recurse(. * .; . < 20)`
Input	`2`

Output	`2`
	`4`
	`16`

Run	

`walk(f)`

The `walk(f)` function applies f recursively to every component of the input entity. When an array is encountered, f is first applied to its elements and then to the array itself; when an object is encountered, f is first applied to all the values and then to the object. In practice, f will usually test the type of its input, as illustrated in the following examples. The first example highlights the usefulness of processing the elements of an array of arrays before processing the array itself. The second example shows how all the keys of all the objects within the input can be considered for alteration.

Examples

Filter	`walk(if type == "array" then sort else . end)`
Input	`[[4, 1, 7], [8, 5, 2], [3, 6, 9]]`
Output	`[[1,4,7],[2,5,8],[3,6,9]]`
Run	

Filter	`walk( if type == "object" then with_entries( .key |= sub( "^_+"; "") ) else . end )`
Input	`[ { "_a": { "__b": 2 } } ]`
Output	`[{"a":{"b":2}}]`
Run	

Stream processing functions

`isempty(expr)`

Returns true if `expr` produces no outputs, false otherwise.

Examples

Filter	`isempty(empty)`
Input	`null`
Output	`true`
Run	

Filter	`isempty(.[])`
Input	`[]`
Output	`true`
Run	

Filter	`isempty(.[])`
Input	`[1,2,3]`
Output	`false`
Run	

`limit(n; expr)`

The `limit` function extracts up to `n` outputs from `expr`.

Examples

Filter	`[limit(3;.[])]`
Input	`[0,1,2,3,4,5,6,7,8,9]`
Output	`[0,1,2]`
Run	

`skip(n; expr)`

The `skip` function skips the first `n` outputs from `expr`.

Examples

Filter	`[skip(3; .[])]`
Input	`[0,1,2,3,4,5,6,7,8,9]`
Output	`[3,4,5,6,7,8,9]`
Run	

`first(expr)`, `last(expr)`, `nth(n; expr)`

The `first(expr)` and `last(expr)` functions extract the first and last values from `expr`, respectively.

The `nth(n; expr)` function extracts the nth value output by `expr`. Note that `nth(n; expr)` doesn't support negative values of `n`.

Examples

Filter	`[first(range(.)), last(range(.)), nth(./2; range(.))]`
Input	`10`
Output	`[0,9,5]`
Run	

`first`, `last`, `nth(n)`

The `first` and `last` functions extract the first and last values from any array at `.`.

The `nth(n)` function extracts the nth value of any array at `.`.

Examples

Filter	`[range(.)]|[first, last, nth(5)]`
Input	`10`
Output	`[0,9,5]`
Run	

JSON conversion functions

`tojson`, `fromjson`

The `tojson` and `fromjson` builtins dump values as JSON texts or parse JSON texts into values, respectively. The `tojson` builtin differs from `tostring` in that `tostring` returns strings unmodified, while `tojson` encodes strings as JSON strings.

Examples

Filter	`[.[]|tostring]`
Input	`[1, "foo", ["foo"]]`
Output	`["1","foo","[\"foo\"]"]`
Run	

Filter	`[.[]|tojson]`
Input	`[1, "foo", ["foo"]]`
Output	`["1","\"foo\"","[\"foo\"]"]`
Run	

Filter	`[.[]|tojson|fromjson]`
Input	`[1, "foo", ["foo"]]`
Output	`[1,"foo",["foo"]]`
Run	

`tostring`

The `tostring` function prints its input as a string. Strings are left unchanged, and all other values are JSON-encoded.

Examples

Filter	`.[] | tostring`
Input	`[1, "1", [1]]`

Output	`"1"`
	`"1"`
	`"[1]"`

Run	

`tonumber`

The `tonumber` function parses its input as a number. It will convert correctly-formatted strings to their numeric equivalent, leave numbers alone, and give an error on all other input.

Examples

Filter	`.[] | tonumber`
Input	`[1, "1"]`
Output	`1` `1`
Run	

Date functions

jq provides some basic date handling functionality, with some high-level and low-level builtins. In all cases these builtins deal exclusively with time in UTC.

High-level date functions

The `fromdateiso8601` builtin parses datetimes in the ISO 8601 format to a number of seconds since the Unix epoch (1970-01-01T00:00:00Z). The `todateiso8601` builtin does the inverse.

The `fromdate` builtin parses datetime strings. Currently `fromdate` only supports ISO 8601 datetime strings, but in the future it will attempt to parse datetime strings in more formats.

The `todate` builtin is an alias for `todateiso8601`.

The `now` builtin outputs the current time, in seconds since the Unix epoch.

Examples

Filter	`fromdate`
Input	`"2015-03-05T23:51:47Z"`
Output	`1425599507`
Run	

Low-level date functions

Low-level jq interfaces to the C-library time functions are also provided: `strptime`, `strftime`, `strflocaltime`, `mktime`, `gmtime`, and `localtime`. Refer to your host operating system's documentation for the format strings used by `strptime` and `strftime`. Note: these are not necessarily stable interfaces in jq, particularly as to their localization functionality.

The `gmtime` builtin consumes a number of seconds since the Unix epoch and outputs a "broken down time" representation of Greenwich Mean Time as an array of numbers representing (in this order): the year, the month (zero-based), the day of the month (one-based), the hour of the day, the minute of the hour, the second of the minute, the day of the week, and the day of the year – all one-based unless otherwise stated. The day of the week number may be wrong on some systems for dates before March 1st 1900, or after December 31 2099.

The `localtime` builtin works like the `gmtime` builtin, but using the local timezone setting.

The `mktime` builtin consumes "broken down time" representations of time output by `gmtime` and `strptime`.

The `strptime(fmt)` builtin parses input strings matching the `fmt` argument. The output is in the "broken down time" representation consumed by `mktime` and output by `gmtime`.

The `strftime(fmt)` builtin formats a time (GMT) with the given format. The `strflocaltime` does the same, but using the local timezone setting.

The format strings for `strptime` and `strftime` are described in typical C library documentation. The format string for ISO 8601 datetime is `"%Y-%m-%dT%H:%M:%SZ"`.

jq may not support some or all of this date functionality on some systems. In particular, the `%u` and `%j` specifiers for `strptime(fmt)` are not supported on macOS.

Compatibility

jaq does not provide any of the given low-level date functions.

Examples

Filter	`strptime("%Y-%m-%dT%H:%M:%SZ")`
Input	`"2015-03-05T23:51:47Z"`
Output	`[2015,2,5,23,51,47,4,63]`
Run	

Filter	`strptime("%Y-%m-%dT%H:%M:%SZ")|mktime`
Input	`"2015-03-05T23:51:47Z"`
Output	`1425599507`
Run	

SQL-style functions

jq provides a few SQL-style functions.

Compatibility

> jaq does not provide any of the functions in this subsection.

`INDEX(stream; index_expression)`

This builtin produces an object whose keys are computed by the given index expression applied to each value from the given stream.

`JOIN($idx; stream; idx_expr; join_expr)`

This builtin joins the values from the given stream to the given index. The index's keys are computed by applying the given index expression to each value from the given stream. An array of the value in the stream and the corresponding value from the index is fed to the given join expression to produce each result.

`JOIN($idx; stream; idx_expr)`

Same as `JOIN($idx; stream; idx_expr; .)`.

`JOIN($idx; idx_expr)`

This builtin joins the input `.` to the given index, applying the given index expression to `.` to compute the index key. The join operation is as described above.

`IN(s)`

This builtin outputs `true` if `.` appears in the given stream, otherwise it outputs `false`.

`IN(source; s)`

This builtin outputs `true` if any value in the source stream appears in the second stream, otherwise it outputs `false`.

Regular expression functions

jq uses the Oniguruma regular expression library, as do PHP, TextMate, Sublime Text, etc, so the description here will focus on jq specifics.

Oniguruma supports several flavors of regular expression, so it is important to know that jq uses the "Perl NG" (Perl with named groups) flavor.

The jq regex filters are defined so that they can be used using one of these patterns:

```
STRING | FILTER(REGEX)
STRING | FILTER(REGEX; FLAGS)
STRING | FILTER([REGEX])
STRING | FILTER([REGEX, FLAGS])
```

where:

- STRING, REGEX, and FLAGS are jq strings and subject to jq string interpolation;
- REGEX, after string interpolation, should be a valid regular expression;
- FILTER is one of `test`, `match`, or `capture`, as described below.

Since REGEX must evaluate to a JSON string, some characters that are needed to form a regular expression must be escaped. For example, the regular expression `\s` signifying a whitespace character would be written as `"\\s"`.

FLAGS is a string consisting of one of more of the supported flags:

- `g` - Global search (find all matches, not just the first)
- `i` - Case insensitive search
- `m` - Multi line mode (`.` will match newlines)
- `n` - Ignore empty matches
- `p` - Both s and m modes are enabled
- `s` - Single line mode (`^` -> `\A`, `$` -> `\Z`)
- `l` - Find longest possible matches
- `x` - Extended regex format (ignore whitespace and comments)

To match a whitespace with the `x` flag, use `\s`, e.g.

```
jq -n '"a b" | test("a\\sb"; "x")'
```

Note that certain flags may also be specified within REGEX, e.g.

```
jq -n '("test", "TEst", "teST", "TEST") | test("(?i)te(?-i)st")'
```

evaluates to `true`, `true`, `false`, `false`.

Compatibility

> jaq uses the `regex` library instead of Oniguruma, which can result in subtle differences in regex execution.

`test(val)`, `test(regex; flags)`

Like `match`, but does not return match objects, only `true` or `false` for whether or not the regex matches the input.

Examples

Filter	`test("foo")`
Input	`"foo"`
Output	`true`
Run	

Filter	`.[] | test("a b c # spaces are ignored"; "ix")`
Input	`["xabcd", "ABC"]`
Output	`true` `true`
Run	

`match(val)`, `match(regex; flags)`

match outputs an object for each match it finds. Matches have the following fields:

- `offset` - offset in UTF-8 codepoints from the beginning of the input
- `length` - length in UTF-8 codepoints of the match
- `string` - the string that it matched
- `captures` - an array of objects representing capturing groups.

Capturing group objects have the following fields:

- `offset` - offset in UTF-8 codepoints from the beginning of the input
- `length` - length in UTF-8 codepoints of this capturing group
- `string` - the string that was captured
- `name` - the name of the capturing group (or `null` if it was unnamed)

Capturing groups that did not match anything return an offset of −1

Examples

Filter	`match("(abc)+"; "g")`
Input	`"abc abc"`
Output	`{"offset": 0, "length": 3, "string": "abc", "captures": [{"offset": 0, "length": 3, "string": "abc", "name": null}]}` `{"offset": 4, "length": 3, "string": "abc", "captures": [{"offset": 4, "length": 3, "string": "abc", "name": null}]}`
Run	

Filter	`match("foo")`
Input	`"foo bar foo"`
Output	`{"offset": 0, "length": 3, "string": "foo", "captures": []}`
Run	

Filter	`match(["foo", "ig"])`
Input	`"foo bar FOO"`
Output	`{"offset": 0, "length": 3, "string": "foo", "captures": []}` `{"offset": 8, "length": 3, "string": "FOO", "captures": []}`
Run	

Filter	`match("foo (?<bar123>bar)? foo"; "ig")`
Input	`"foo bar foo foo  foo"`
Output	`{"offset": 0, "length": 11, "string": "foo bar foo", "captures": [{"offset": 4, "length": 3, "string": "bar", "name": "bar123"}]}` `{"offset": 12, "length": 8, "string": "foo  foo", "captures": [{"offset": -1, "length": 0, "string": null, "name": "bar123"}]}`
Run	

Filter	`[ match("."; "g")] | length`
Input	`"abc"`
Output	`3`
Run	

`capture(val)`, `capture(regex; flags)`

Collects the named captures in a JSON object, with the name of each capture as the key, and the matched string as the corresponding value.

Examples

Filter	`capture("(?<a>[a-z]+)-(?<n>[0-9]+)")`
Input	`"xyzzy-14"`
Output	`{ "a": "xyzzy", "n": "14" }`
Run	

`scan(regex)`, `scan(regex; flags)`

Emit a stream of the non-overlapping substrings of the input that match the regex in accordance with the flags, if any have been specified. If there is no match, the stream is empty. To capture all the matches for each input string, use the idiom `[ expr ]`, e.g. `[ scan(regex) ]`.

Examples

Filter	`scan("c")`
Input	`"abcdefabc"`
Output	`"c"` `"c"`
Run	

`split(regex; flags)`

Splits an input string on each regex match.

For backwards compatibility, when called with a single argument, `split` splits on a string, not a regex.

Examples

Filter	`split(", *"; null)`
Input	`"ab,cd, ef"`
Output	`["ab","cd","ef"]`
Run	

`splits(regex)`, `splits(regex; flags)`

These provide the same results as their `split` counterparts, but as a stream instead of an array.

Examples

Filter	`splits(", *")`
Input	`"ab,cd,   ef, gh"`

Output	`"ab"`
	`"cd"`
	`"ef"`
	`"gh"`

Run	

`sub(regex; tostring)`, `sub(regex; tostring; flags)`

Emit the string obtained by replacing the first match of regex in the input string with `tostring`, after interpolation. `tostring` should be a jq string or a stream of such strings, each of which may contain references to named captures. The named captures are, in effect, presented as a JSON object (as constructed by `capture`) to `tostring`, so a reference to a captured variable named "x" would take the form: `"\(.x)"`.

Examples

Filter	`sub("[^a-z]*(?<x>[a-z]+)"; "Z\(.x)"; "g")`
Input	`"123abc456def"`
Output	`"ZabcZdef"`
Run	

Filter	`[sub("(?<a>.)"; "\(.a|ascii_upcase)", "\(.a|ascii_downcase)")]`
Input	`"aB"`
Output	`["AB","aB"]`
Run	

`gsub(regex; tostring)`, `gsub(regex; tostring; flags)`

`gsub` is like `sub` but all the non-overlapping occurrences of the regex are replaced by `tostring`, after interpolation. If the second argument is a stream of jq strings, then `gsub` will produce a corresponding stream of JSON strings.

Examples

Filter	`gsub("(?<x>.)[^a]*"; "+\(.x)-")`
Input	`"Abcabc"`
Output	`"+A-+a-"`
Run	

Filter	`[gsub("p"; "a", "b")]`
Input	`"p"`
Output	`["a","b"]`
Run	

I/O functions

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`, `inputs`

The filter `input` outputs one new input, and the filter `inputs` outputs all remaining inputs. This is primarily useful for reductions over a program's inputs.

Note

When using `input` or `inputs`, it is often necessary to invoke jq with the `-n` command-line option to avoid losing the first value in the input stream.

```
$ echo 1 2 3 4 | jq '[., input]'
[1,2]
[3,4]
$ echo 1 2 3 | jq -n 'reduce inputs as $i (0; . + $i)'
6
```

Compatibility

When there is no more input value left, in jq, `input` yields an error, whereas in jaq, `input` yields no output value, i.e. `empty`.

`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.

`halt`

Stops the jq program with no further outputs. jq will exit with exit status `0`.

`halt_error`, `halt_error(exit_code)`

Stops the jq program with no further outputs. The input will be printed on `stderr` as raw output (i.e., strings will not have double quotes) with no decoration, not even a newline.

The given `exit_code` (defaulting to `5`) will be jq's exit status.

For example, `"Error: something went wrong\n"|halt_error(1)`.

`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.

Compatibility

jaq does not provide this function.

`input_line_number`

Returns the line number of the input currently being filtered.

Compatibility

> jaq does not provide this function.

`$ENV`, `env`

`$ENV` is an object representing the environment variables as set when the jq program started.

`env` outputs an object representing jq's current environment.

At the moment there is no builtin for setting environment variables.

Examples

Filter	`$ENV.PAGER`
Input	`null`
Output	`"less"`
Run	

Filter	`env.PAGER`
Input	`null`
Output	`"less"`
Run	

Streaming functions

With the `--stream` option jq can parse input texts in a streaming fashion, allowing jq programs to start processing large JSON texts immediately rather than after the parse completes. If you have a single JSON text that is 1GB in size, streaming it will allow you to process it much more quickly.

However, streaming isn't easy to deal with as the jq program will have `[<path>, <leaf-value>]` (and a few other forms) as inputs.

Several builtins are provided to make handling streams easier.

The examples below use the streamed form of `[0,[1]]`, which is `[[0],0],[[1,0],1],[[1,0]],[[1]]`.

Streaming forms include `[<path>, <leaf-value>]` (to indicate any scalar value, empty array, or empty object), and `[<path>]` (to indicate the end of an array or object). Future versions of jq run with `--stream` and `--seq` may output additional forms such as `["error message"]` when an input text fails to parse.

Compatibility

> Because jaq does not support the `--stream` option, it does not provide any of the functions in this subsection.

`truncate_stream(stream_expression)`

Consumes a number as input and truncates the corresponding number of path elements from the left of the outputs of the given streaming expression.

Examples

Filter	`truncate_stream([[0],1],[[1,0],2],[[1,0]],[[1]])`
Input	`1`

Output	`[[0],2]`
	`[[0]]`

Run	

`fromstream(stream_expression)`

Outputs values corresponding to the stream expression's outputs.

Examples

Filter	`fromstream(1|truncate_stream([[0],1],[[1,0],2],[[1,0]],[[1]]))`
Input	`null`
Output	`[2]`
Run	

`tostream`

The `tostream` builtin outputs the streamed form of its input.

Examples

Filter	`. as $dot\|fromstream($dot|tostream)|.==$dot`
Input	`[0,[1,{"a":1},{"b":2}]]`
Output	`true`
Run	

Miscellaneous

Compatibility

jaq does not provide any of the symbols in this subsection.

`modulemeta`

Takes a module name as input and outputs the module's metadata as an object, with the module's imports (including metadata) as an array value for the `deps` key and the module's defined functions as an array value for the `defs` key.

Programs can use this to query a module's metadata, which they could then use to, for example, search for, download, and install missing dependencies.

`$__loc__`

Produces an object with a "file" key and a "line" key, with the filename and line number where `$__loc__` occurs, as values.

Examples

Filter	`try error("\($__loc__)") catch .`
Input	`null`
Output	`"{\"file\":\"<top-level>\",\"line\":1}"`
Run	

`builtins`
Returns a list of all builtin functions in the format `name/arity`. Since functions with the same name but different arities are considered separate functions, `all/0`, `all/1`, and `all/2` would all be present in the list.

`have_literal_numbers`
This builtin returns true if jq's build configuration includes support for preservation of input number literals.

`have_decnum`
This builtin returns true if jq was built with "decnum", which is the current literal number preserving numeric backend implementation for jq.

The examples below use the builtin function `have_decnum` in order to demonstrate the expected effects of using / not using `--disable-decnum`. They also allow automated tests derived from these examples to pass regardless of whether that option is used.

Examples

Filter	`[., tojson] | . == if have_decnum then [12345678909876543212345,"12345678909876543212345"] else [12345678909876543000000,"12345678909876543000000"] end`
Input	`12345678909876543212345`
Output	`true`
Run	

Filter	`map([., . == 1]) | tojson | . == if have_decnum then "[[1,true],[1.000,true],[1.0,true],[1.00,true]]" else "[[1,true],[1,true],[1,true],[1,true]]" end`
Input	`[1, 1.000, 1.0, 100e-2]`
Output	`true`
Run	

Filter	`. as $big \| [$big, $big + 1] | map(. > 10000000000000000000000000000000) | . == if have_decnum then [true, false] else [false, false] end`
Input	`10000000000000000000000000000001`
Output	`true`
Run	

`$JQ_BUILD_CONFIGURATION`
This builtin variable shows the jq executable's build configuration. Its value has no particular format, but it can be expected to be at least the `./configure` command-line arguments, and may be enriched in the future to include the version strings for the build tooling used.

Note that this can be overridden in the command-line with `--arg` and related options.