JQ Distilled

JQ programs consume a stream of JSON values and process them with one or more combined filters. The input may also consist on a stream of UTF-8 lines (like the output) or on a single big string. Filters are parametrized generators that consume one input JSON value and produce a stream of output JSON values.

JSON values

object
  {}
  { *members* **}**
members
  pair
  pair **,** *members*
pair
  string **:** *value*
array
  []
  [*elements* **]**
elements
  value
  value **,** *elements*

value
  string
  number
  object
  array
  true
  false
  null

---

string
  ""
  " *chars* **"**
chars
  char
  char chars

char
  any Unicode character except **"**
    or **** or control character
  \"
  \\ \/
  \b \f
  \n \r \t
  \u*four-hex-digits*
number
  int
  int frac
  int exp
  int frac exp

int
  digit
  digit1-9 digits
  - *digit*
  - *digit1-9 digits*
frac
  . *digits*
exp
  e digits
digits
  digit
  digit digits
e
  e e+ e- E E+ E-

The constants **`null`**, **`false`** and **`true`**, number and string literals and array and object constructors define JSON values. JQ extends JSON with the numeric constants **`nan`** and **`infinite`** (and input literals `NaN` and `Inf`). Object constructors offer several syntactic extensions to JSON literals:

```
{foo: bar}        = {"foo": bar}
{foo}             = {"foo": .foo}
{$foo}            = {"foo": $foo}
{("fo"+"o"): bar} = {"foo": bar}
```

JQ evaluation model is better understood adding the operational *values* `@` (the empty stream) and **`!`** (cancel symbol). New filters are built using operators and special constructs. In increasing order of priority the operators are:

Operator	Assoc.	Description
(…)		scope delimiter and grouping operator
`|`	right	sequence two filters; succeeds if both operands succeed
`,`	left	alternates two filters; succeeds if any operand succeed
`//`	right	coerces **`null`**, **`false`** and `@` to an alternative value
`= |= += -= *= /= %= //=`	nonassoc	assign, update
`or`	left	boolean "or"
`and`	left	boolean "and"
`!= == < > <= >=`	nonassoc	boolean tests
`+ -`	left	polymorphic plus and minus
`* / %`	left	polymorphic multiply and divide; modulo
`-`	none	prefix negation
?	none	postfix operator, coerces **`!`** (if cacheable) to `@`

JQ defines the following complete order for JSON values, including **`nan`** and **`infinite`**:

```
null < false < true < nan < -(infinite) < numbers < infinite < strings < arrays < objects
```

The **`as`** construct binds variables names and supports array and object destructuring. Binding of variables and sequencing and alternation of filters can be described with the following equivalences:

$$(a_1,a_2,\ldots,a_n)\ \texttt{as \$a}\ \texttt{|}\ f(\texttt{\$a}) \equiv (f(a_1),f(a_2),\ldots,f(a_n))$$
$$(a_1,a_2,\ldots,a_n)\ \texttt{|}\ f \equiv (a_1\texttt{|}f,a_2\texttt{|}f,\ldots,a_n\texttt{|}f)$$
$$(a_1,a_2,\ldots,a_n)\ \texttt{,}\ (b_1,b_2,\ldots,b_n) \equiv (a_1,a_2,\ldots,a_n,b_1,b_2,\ldots,b_n)$$

Control flow is organized with the operators **|**, **,** and the constructs **if**, **reduce**, **foreach**, **label** and **try**. The postfix **?** operator is syntactic sugar for the **try** special construct.

Schematic syntax for special constructs

```
def name: expression;
def name(parameters): expression;
term as pattern | expression
if expression then expression else expression end
if expression then expr elif expr then expr ... else expr end
reduce term as pattern (init; update)     # init, update and extract are expressions
foreach term as pattern (init; update)
foreach term as pattern (init; update; extract)
label $name | expression ... break $name
try expression
try expression catch expression
```

New filters can be defined with the **def** construct. Filters consume one input value, receive zero or more parameters and produce zero or more output values. Parameters can be passed by name, or by value if prefixed with the character **$** in the filter definition.

Core predefined filters

Filter	Description
`.`	produces unchanged its input value; is the identity filter; always succeeds
`empty`	does not produce any value on its output (produces `@`); never succeeds
`.k ."k"`	object member access; shorthand for **`.["k"]`**
`x[k]`	array element and object member access
`x[i:j]`	array or string slice
`[]`	generates objects and arrays values
`..`	Recursively descends `.`, producing **`.,.[]?,(.[]?|.[]?),...`**
`keys`	generates ordered array indices and object keys
`length`	size of strings, arrays and objects; absolute value of numbers
`del(path)`	removes ***`path`*** in the input value
`type`	produces as string the type name of JSON values
`explode, implode`	conversion of strings to/from code point arrays
`tojson, fromjson`	conversion of JSON values to/from strings
`"\(expr)"`	string interpolation
`@fmt`	format and escape strings
`error, error(value)`	signals an error canceling the current filter (produces **`!`**); can be caught
`halt, halt_error(status)`	signals an error exiting the program

After parameter instantiation JQ filters are like binary relations on JSON values and follow several algebraic laws (in the following table > means `select` and `length/1` is assumed to be defined for streams):

`. | A ≡ A ≡ A | .`
`@ | A ≡ @ ≡ A | @`
`A | ! | B ≡ !`
	`length(`a_1`,`a_2`,…,`a_n`) = `n
	`length(range(`m`;`n`)) = `$n-m$
	`length(empty) = 0`

`@ , A ≡ A ≡ A , @` `A , ! , B ≡ A , !`	`A , (B , C) ≡ (A , B) , C` `A | (B | C) ≡ (A | B) | C`

`(A , B) | >(p) ≡ (A | >(p)) , (B | >(p))`
`A | >(p) | >(q) ≡ A | >(q) | >(p)`
`A | >(p) | >(p) ≡ A | >(p)`
`A | B | >(p) ≡ A | >(B | p) | B`
	`(A , B) | C ≡ (A | C) , (B | C)`

JQ has a dynamic type system but, to better describe filters behavior, type signatures can be added as comments.

Proposed grammar for filters type signatures

type anotation
  :: *places*
places
  output
  => *output*
  *input***| =>** *output*
  (*parameters***) =>** *output*
  *input***| (***parameters***) =>** *output*
parameters
  parameter
  *parameter***;** *parameters*
input
  value
	parameter
	  value
	  *value***->***stream*[1]
	output
	  stream
	  ![2]
	stream
	  @[3]
	  value
	  ?*value*[4]
	  *****value
	  +*value*
	  *stream***!**[5]
		value
		  null
		  boolean
		  number
		  string
		  array
		  object
		  [*value***]**
		  {*value***}**
		  <*value***>**[6]
		  *value***^***value*[7]
		  letter[8]
		  name[9]

Notes:

[1] Parameters passed by name are like parameterless filters.

[2] The character **!** is the display symbol for non-terminating filters type.

[3] The character **@** denotes the empty stream. Use only when no results are expected.

[4] Occurrence indicators (**?**, *****, **+**) have the usual meaning.

[5] Streams output type always have an implicit union with **!**. To add only when cancellation is expected.

[6] Indistinct array or object: **<a>** ≡ **[a]^{a}**.

[7] Union of two value types.

[8] Single lowercase letters are type variables representing indeterminate JSON value types.

[9] Named object (construct only with the underscore character and uppercase letters).

Type anotations examples

```
add/0 :: <number^string^array^object>| => null^number^string^array^object
all/0 :: [boolean]| => boolean
all/1 :: [a]|(a->boolean) => boolean
all/2 :: a|(a->*b;b->boolean) => boolean
arrays/0 :: a| => ?array
ascii_downcase/0 :: string| => string
booleans/0 :: a| => ?boolean
bsearch/1 :: [a]|(a) => number
contains/1 :: a|(a) => boolean
del/1 :: object|(PATH) => object
delpaths/1 :: object|([[number^string]]) => object
empty/0 :: => @
endswith/1 :: string|(string) => boolean
env/0 :: => {string}
error/1 :: (a) => !
error/0 :: => !
explode/0 :: string| => [number]
finites/0 :: a| => ?number
first/0 :: [a]| => a
first/1 :: a|(a->*b) => ?b
floor/0 :: number| => number
has/1 :: object|(string) => boolean
implode :: [number]| => string
```