

JQ Distilled

JQ process streams of JSON values using one or more combined filters. The input may also consist on a stream of UTF-8 lines or a single big UTF-8 string. Filters are parameterized generators that consume JSON values and produce a stream of JSON values.

JSON values

<i>object</i> { } { members } <i>members</i> <i>pair</i> <i>pair , members</i> <i>pair</i> <i>string : value</i> <i>array</i> [] [elements] <i>elements</i> <i>value</i> <i>value , elements</i>	<i>value</i> <i>string</i> <i>number</i> <i>object</i> <i>array</i> true false null <hr/> <i>string</i> " " " chars " <i>chars</i> <i>char</i> <i>char chars</i>	<i>char</i> <i>any Unicode character except " or \ or control character</i> \ " \ \ \ / \ b \ f \ n \ r \ t \ u four-hex-digits <i>number</i> <i>int</i> <i>int frac</i> <i>int exp</i> <i>int frac exp</i>	<i>int</i> <i>digit</i> <i>digit1-9 digits</i> <i>- digit</i> <i>- digit1-9 digits</i> <i>frac</i> <i>. digits</i> <i>exp</i> <i>e digits</i> <i>digits</i> <i>digit</i> <i>digit digits</i> <i>e</i> 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 adds the numeric constants **nan** and **infinite** in JQ scripts, and literals **NaN** and **Inf** in JSON input data. Object constructors offer several syntactic extensions respect 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 non assignable values **@** (the *empty stream*) and **!** (the *abort 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	relational tests
+ -	left	polymorphic plus and minus
* / %	left	polymorphic multiply and divide; modulo
-	none	prefix negation
?	none	postfix operator, coerces ! 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 variable names and supports array and object destructuring. Binding of variables and sequencing and alternation of filters can be described with the following equivalences:

```
(a1, a2, ..., an) as $a | f($a)  ≡ f(a1), f(a2), ..., f(an)
(a1, a2, ..., an) | f              ≡ (a1 | f), (a2 | f), ..., (an | f)
(a1, a2, ..., an), (b1, b2, ..., bm) ≡ a1, a2, ..., an, b1, b2, ..., bm
```

The special constructs **if**, **reduce**, **foreach**, **label** and **try** extend JQ control flow capabilities. 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 expr else expr 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 are passed by name, or by value if prefixed with the character **\$** in the filter definition.

Core predefined filters

Filter	Description
.	identity filter, produces unchanged its input value; always succeed
empty	does not produce any value on its output (<i>produces @</i>); always fail
null false	boolean "false"
true	boolean "true", as everything else except <code>null</code> and <code>false</code>
.k ."k"	object identifier-index; shorthand for <code>["k"]</code>
x[k]	array index and generic object index
x[i:j]	array and string slice
x[]	array and object value iterator
..	recursively descends <code>..</code> , producing <code>.. []?</code> , <code>(.. []?) .. []?</code> , ...
keys	generates ordered array indices and object keys
length	size of strings, arrays and objects; absolute value of numbers
del(path)	removes <i>path</i> 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 aborting the current filter (<i>produces !</i>); can be caught
halt, halt_error(status)	exits the program

After parameter instantiation JQ filters are like binary relations on JSON values, and follow several algebraic laws (in the following table stands for `select/1`):

$@, A \equiv A \equiv A, @$ $. A \equiv A \equiv A .$ $@ A \equiv @ \equiv A @$	$A, (B, C) \equiv (A, B), C$ $A (B C) \equiv (A B) C$ $(A, B) C \equiv (A C), (B C)$ $A (B, C) \equiv (A B), (A C)$
$(A, B) s(p) \equiv (A s(p)), (B s(p))$ $s(p) s(q) \equiv s(q) s(p)$ $s(p) s(p) \equiv s(p)$ $A B s(p) \equiv A s(B p) B$	$A ! \equiv !$ $A, !, B \equiv A, !$

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

Proposed grammar for filter type signatures

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

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

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

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

⁷ Union of two value types.

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

⁹ Named object (use only the underscore character and uppercase letters).

