JQ programs consume a stream of JSON values processing 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 subroutines 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 denote JSON values. JQ extends JSON with the numeric constants **nan** and **infinite**, and the operational values ⌀ and ⊥. Object constructors offer several syntactic extensions to JSON literals:

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

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 ⊥ 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 pseudocode:

```
A as $a | f($a) = foreach A as $a (f($a))  # applies f to A's output in a loop
(A | B) = foreach A as $a (B[.=$a])  # applies B with . replaced by A's output
(A , B) = foreach A as $a ($a) , foreach B as $b ($b) # join streams
```

Evaluation 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 receive zero or more parameters, consume one input value and produce zero (⌀) or more output values. Parameters can be passed by name, or by value if prefixed with the character **`$`**. Canceled filters produce then ⊥ value.

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; never succeeds; produces ⌀
`.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 cancelling the current filter; produces ⊥ (can be catched)
`halt, halt_error(status)`	signals an error exiting the program; produces ⊥

After parameter instantiation JQ filters are binary relations on JSON values.

JQ algebraic laws

`. | A ≡ A` `A | . ≡ A`	`⌀ , A ≡ A` `A , ⌀ ≡ A`
`⌀ | A ≡ ⌀` `A | ⌀ ≡ ⌀`	`A , (B , C) ≡ (A , B) , C` `A | (B | C) ≡ (A | B) | C`
`A , ⊥ , B ≡ A , ⊥` `A | ⊥ | B ≡ ⊥`	`(A , B) | C ≡ (A | C) , (B | C)` `A`[1] `| (B , C) ≡ (A | B) , (A | C)`

1. If `A` cancels left-associativity is not satisfied.

JQ has a dynamic type system but, to help understand filters behavior, type annotations can added inside comments.

Grammar for JQ filters type anotations

type anotation
  #:: *places*
places
  output
  => *output*
  input **| =>** *output*
  (*parameters***) =>** *output*
  input **|** *(parameters)* **=>** *output*
parameters
  parameter
  parameter **;** *parameters*
	input
	  type
	parameter
	  type
	output
	  type
	  *type***^⊥**[1]
	type
	  name
	  stream
	  value
	  ⊥[2]
	name[3]
	  value **->** *value*
	  value **->** *stream*
		stream
		  ∅[4]
		  ? *value*[5]
		  ***** *value*
		  + *value*
		value
		  null
		  boolean
		  number
		  string
		  array
		  object
		  [*value***]**
		  {*value***}**
		  <*value***>**[6]
		  a..z[7]
		  [A-Z_]+[8]
		  *value***^***value*[9]

[1] Output types have always an implicit union with ⊥. To be added explicitly only when cancellation is expected.
[2] Bottom type (type with no values). Denote the value for filters that cancel.
[3] Parameters passed by name are like parameterless filters.
[4] Empty stream.
[5] Occurrence indicators (**?**, *****, **+**) have the regular expressions usual meaning.
[6] Indistinct arrays or objects.
[7] Indeterminate JSON value type (it's not necessary to know the exact type).
[8] Named object.
[9] Union of two types.

Types allowed in each place

	input	*parameter*	*output*
value	✔	✔	✔
stream			✔
name		✔	
⊥			✔