jq distilled

This text tries to be the briefest possible description of the essential characteristics of the *jq* language; it is therefore incomplete by definition. It should also be noted that the author's mother tongue is not English, and that any help received to improve the wording of the text will be well received.

The `jq` command-line processor transforms streams of input JSON values using one or more combined filters written in the *jq* language. The input may also consist on UTF-8 text lines or a single big UTF-8 string. Filters are parameterized generators that for each consumed JSON value produce a stream of zero or more 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-**

In the *jq* language the constants **null**, **false** and **true**, number and string literals and array and object constructors define JSON values; no other kind of values exists. *jq* adds the numeric constants **nan** and **infinite**, and also accepts as an extension the 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 two non assignable "values" denoted by **@** (the *empty stream*) and **!** (the *non-termination 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	compose/sequence two filters
`,`	left	concatenate/alternate two filters
`//`	right	coerces **null**, **false** and **@** to an alternative value
`= |= += -= *= /= %= //=`	nonassoc	assign; update
`or`	left	boolean "or"
`and`	left	boolean "and"
`== != < > <= >=`	nonassoc	equivalence and precedence tests
`+ -`	left	polymorphic plus and minus
`* / %`	left	polymorphic multiply and divide; modulo
`-`	none	prefix negation
`?`	none	postfix operator, coerces **!** to **@**
`?//`	nonassoc	destructuring alternative operator

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:

$$\begin{aligned}
(a_1,a_2,\ldots,a\ )\ \texttt{as \$a}\ |\ f(\texttt{\$a}) &\equiv f(a_1),f(a_2),\ldots,f(a\ )\\
(a_1,a_2,\ldots,a\ )\ |\ f &\equiv (a_1|f),(a_2|f),\ldots,(a\ |f)\\
(a_1,a_2,\ldots,a\ )\ ,\ (b_1,b_2,\ldots,b\ ) &\equiv a_1,a_2,\ldots,a\ ,b_1,b_2,\ldots,b
\end{aligned}$$

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 { ?// pattern }| expression
if expression then expr end
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 expr.
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, can have extra 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
`empty`	empty filter, does not produce any value on its output (*produces* `@`)
`null false`	boolean "false"
`true`	boolean "true", as everything else except **`null`** and **`false`**
`.k ."k"`	object identifier-index; shorthand for **`.["k"]`**
`x[k]`	array index and generic object index
`x[i:j]`	array and string slice
`x[]`	array and object value iterator
`..`	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 aborting the current filter (*produces* **`!`**); can be caught
`halt, halt_error(status)`	exits the program

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

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

`(A , B) | ^(p) ≡ (A | ^(p)) , (B | ^(p))`
`^(p) | ^(q)    ≡ ^(q) | ^(p)`
`^(p) | ^(p)    ≡ ^(p)`
`A | B | ^(p)   ≡ A | ^(B | p)`
	`! | A  ≡  !  ≡  A | !`

jq has a dynamic type system but, to better describe filters behavior in scripts, is advisable to add type signatures as comments.

Proposed grammar for filter type signatures

type annotation	*parameter*	*value*
`::` *places*	*value*	**`null`**
places	*value*`->`*stream*[1]	**`boolean`**
output	*output*	**`number`**
`=>` *output*	*stream*	**`string`**
input **`=>`** *output*	**`!`**[2]	**`array`**
`(`*parameters*`)` **`=>`** *output*	*stream*	**`object`**
input`|(`*parameters*`)` **`=>`** *output*	`@`[3]	`[`*value*`]`
parameters	*value*	`{`*value*`}`
parameter	`?`*value*[4]	`<`*value*`>`[6]
parameter`;` *parameters*	`*`*value*	*value*`^`*value*[7]
input	`+`*value*	*letter*[8]
value	*stream*`!`[5]	*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.
[4] Occurrence indicators (`?, *, +`) have the usual meaning.
[5] Streams output type always have an implicit union with `!`. To add only when non-termination 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 (use only the underscore character and uppercase letters).