CMPT 473 Assignment 2 Test Report

1. Introduction & Program Under Test

Program: `jq` a lightweight, portable command-line JSON processor **Version tested:** jq-1.7.1-apple (macOS), jq-1.6 (macOS), jq-1.8.1 (Windows) **Specification:** jq Manual **Source:** https://github.com/jqlang/jq

`jq` reads JSON input (from a file or stdin), applies a filter expression, and writes the result to stdout. It supports a wide range of input modes, output formatting options, transformation filters, and error/exit-code behaviors.

Team Contributions

Split	Owner	Scope
Split 1	Jeffrey Wong (`jwa334`)	Input modes & file ingestion (`-R` , `-s` , `-n` , file vs stdin)
Split 2	Tommy Duong (`tda49`)	Output formatting & encoding (`-c` , `-r` , `-j` , `--raw-output0`)
Split 3	Dean Zhou (`dza68`)	Error handling & exit status (`-e` , parse/compile/file errors)
Split 4	Ronney Lok (`rla121`)	Transform/computation filters (`select` , `map` , `sort` , `group_by` , `unique` , `add` , `length`)

2. Input Space Model

2.1 Split 1 Input Modes & File Ingestion

Specification reference: jq Manual: "Invoking jq", options `-R` , `-s` , `-n`

Parameters & Categories

Parameter	Choices
`input_source`	`file` , `stdin`
`input_content`	`valid_json` , `multi_json` , `plain_text` , `empty` , `invalid_json` , `missing_file`
`raw_input`	`off` , `on`
`slurp`	`off` , `on`
`null_input`	`off` , `on`

Constraints

ID	Constraint	Rationale
C1	`input_content = missing_file => input_source = file`	A missing-file error can only occur with a file path argument, not stdin
C2	`null_input = on => raw_input = off AND slurp = off`	`-n` ignores all input; `-R` / `-s` have no effect when combined with it
C3	`null_input = on => input_content in {valid_json, empty}`	Content is irrelevant when ignored; avoids meaningless frames
C4	`input_content = missing_file => raw_input = off AND slurp = off AND null_input = off`	jq fails immediately on open error before input flags apply
C5	`input_content = invalid_json AND raw_input = off => parse error expected`	`-R` bypasses JSON parsing; without it, invalid JSON triggers a parse error

2.2 Split 2 Output Formatting & Encoding

Specification reference: jq manual: "Basic filters", options `-c` , `-r` , `-j` , `--raw-output0`

Parameters & Categories

Parameter	Choices
`output_mode`	`default` , `compact` , `raw` , `join` , `raw0`
`output_type`	`string` , `number` , `object` , `array` , `bool` , `null`
`string_content_class`	`plain` , `quote_backslash` , `unicode` , `embedded_newline` , `not_applicable`
`result_cardinality`	`single` , `multiple`

Constraints

ID	Constraint	Rationale
C1	`string_content_class != not_applicable => output_type = string`	Content class only meaningful for string outputs
C2	`output_type != string => string_content_class = not_applicable`	Non-string types have no content class
C3	`output_type = string => string_content_class != not_applicable`	String outputs must specify a content class
C4	`output_mode in {raw, join, raw0} => output_type = string`	Raw/join/raw0 modes are only defined for string outputs

2.3 Split 3 Error Handling & Exit Status

Specification reference: jq manual: "Invoking jq", option `-e / --exit-status` ; jq exit code table

Parameters & Categories

Parameter	Choices
`input_file_state`	`valid` , `invalid_json` , `empty` , `missing` , `no_read_permission`
`filter_kind`	`dot` , `invalid_program` , `field_a` , `field_b` , `field_c` , `field_missing` , `empty_filter`
`exit_status_flag`	`off` , `on`

Constraints

ID	Constraint	Rationale
C1	`input_file_state != valid => filter_kind = dot`	Ensures test outcomes (exit/stderr) reflect only file/parse/open errors
C2	`input_file_state != valid => exit_status_flag = off`	`-e` behavior is only meaningful when a valid result is produced
C3	`filter_kind = invalid_program => input_file_state = valid`	Compile errors require a valid, readable input file
C4	`filter_kind in {field_a, field_b, field_c, field_missing, empty_filter} => input_file_state = valid AND exit_status_flag = on`	`-e` exit-code semantics are tested only on valid input with `-e` enabled

2.4 Split 4 Transform/Computation Filters

Specification reference: jq manual: "Builtin operators and functions" (`select` , `map` , `sort` , `group_by` , `unique` , `add` , `length`)

Parameters & Categories

Parameter	Choices
`filter_type`	`select` , `map` , `sort` , `group_by` , `unique` , `add` , `length`
`input_data_type`	`array_numeric` , `array_object` , `array_mixed` , `array_empty` , `scalar_string` , `scalar_number` , `object_simple` , `null`
`data_property`	`sorted` , `unsorted` , `has_duplicates` , `missing_keys`

Constraints

ID	Constraint	Rationale
C1	`filter_type in {sort, group_by, unique, add, map, select} => input_data_type in {array_*, object_simple}`	These filters operate on arrays or object_simple; scalars and null are out of scope
C2	`filter_type in {sort, group_by, unique} => data_property in {sorted, unsorted, has_duplicates}`	Behavior only applies to sortable data
C3	`data_property = missing_keys => input_data_type in {array_object, object_simple}`	Missing-key behavior applies to object arrays or simple objects
C4	`filter_type = length => input_data_type in {array_*, scalar_string, object_simple, null}`	Length is defined for these types; scalar_number excluded
C5	`filter_type = add => input_data_type in {array_*}`	`add` reduces arrays; scalar reduction is out of scope

3. Test Frame Generation

Method

Test frames were generated using the **category-partition method** and **combinatorial (pairwise / 2-way) coverage**. Splits 3 and 4 used the **ACTS 3.0 tool** (jar at `__MACOSX/ACTS3.0/acts_3.0.jar`) to automatically generate frames from their ACTS input models. Example command (split 3):

```
java -Dalgo=ipog -Ddoi=2 -Doutput=csv -Dmode=scratch \
    -Dchandler=forbiddentuples -Dcheck=on \
    -jar __MACOSX/ACTS3.0/acts_3.0.jar \
    docs/acts/split3/split3_acts_input.txt \
    docs/acts/split3/split3_pairwise_frames.csv
```

Frame Counts

Split	Parameters	Constraints	ACTS Frames Generated	Tests Implemented	Frame File
Split 1	5	5	18	18	`docs/acts/split1/split1_pairwise_frames.csv`
Split 2	4	4	30	30	`docs/acts/split2/split2_pairwise_frames.csv`
Split 3	3	4	12	12	`docs/acts/split3/split3_pairwise_frames.csv`
Split 4	3	7	38	15	`docs/acts/split4/split4_pairwise_frames.csv`

4. Test Case Generation & Results

Method

Each pairwise frame maps directly to one pytest test function. Tests invoke `jq` by the shared helper `tests/common/run_jq.py` and assert on:

- **Return code** (`rc`): exact value or nonzero class
- **stdout**: JSON structural equality (`json.loads`) or exact byte comparison
- **stderr**: empty, or pattern match (e.g., `parse`, `compile|syntax`, `could not open`)

4.1 Split 1 Results

Frame	Test	rc	stdout	stderr	Result
F01	`test_file_valid_json_default`	0	JSON object	empty	Pass
F02	`test_stdin_valid_json_default`	0	JSON object	empty	Pass
F03	`test_file_multi_json_default`	0	2 JSON values	empty	Pass
F04	`test_stdin_multi_json_default`	0	2 JSON values	empty	Pass
F05	`test_file_plain_text_raw_input`	0	quoted lines	empty	Pass
F06	`test_stdin_plain_text_raw_input`	0	quoted lines	empty	Pass
F07	`test_file_valid_json_slurp`	0	array of 1	empty	Pass
F08	`test_stdin_multi_json_slurp`	0	array of 2	empty	Pass
F09	`test_file_plain_text_raw_slurp`	0	concatenated string	empty	Pass
F10	`test_stdin_valid_json_raw_slurp`	0	concatenated string	empty	Pass
F11	`test_file_null_input`	0	null	empty	Pass
F12	`test_stdin_null_input_empty`	0	null	empty	Pass
F13	`test_file_missing_error`	2	empty	file-open error	Pass
F14	`test_file_invalid_json_parse_error`	nonzero	empty	parse error	Pass
F15	`test_stdin_invalid_json_parse_error`	nonzero	empty	parse error	Pass
F16	`test_file_invalid_json_raw_input_bypasses_parse`	0	raw string	empty	Pass
F17	`test_file_empty_default`	0	empty	empty	Pass
F18	`test_file_empty_slurp`	0	`[]`	empty	Pass

18/18 passed.

4.2 Split 2 Results

All 30 frames executed by the parameterized test `test_split2_frame[F01]` – `test_split2_frame[F30]`. Each frame asserts exact stdout byte output (including newline/NUL terminators) and empty stderr.

Frames	Mode	Output types covered	Result
F01–F10	`default`, `compact`	string (all 4 classes), number, object, array, bool, null	Pass
F11–F18	`raw`, `join`, `raw0`, `compact`	string (all classes), bool, null, array	Pass
F19–F30	`default`, `compact`, `raw`, `join`, `raw0`	string (all 4 classes), single and multiple cardinality	Pass

30/30 passed.

4.3 Split 3 Results

Frame	Test	Expected rc	Expected stderr	Result
F01	`test_invalid_json_file_returns_parse_error_and_no_stdout`	nonzero	`parse`	Pass
F02	`test_empty_json_file_returns_parse_error_and_no_stdout`	0 or nonzero	`parse` or empty	Pass
F03	`test_invalid_filter_returns_compile_or_syntax_error_and_no_stdout`	3	`compile|syntax`	Pass
F04	`test_missing_file_returns_open_error_and_no_stdout`	2	`could not open|no such file`	Pass
F05	`test_permission_denied_returns_error`	2	`permission|denied`	Pass
F06	`test_exit_status_truthy_output_returns_zero_nonempty_stdout_empty_stderr`	0	empty	Pass
F07	`test_exit_status_missing_returns_one_and_quiet_stderr`	1	empty	Pass
F08	`test_exit_status_false_returns_one`	1	empty	Pass
F09	`test_exit_status_null_returns_one`	1	empty	Pass
F10	`test_exit_status_no_output_empty_filter_returns_four`	4	empty	Pass
F11	`test_exit_status_invalid_filter_returns_three_and_compile_error`	3	`compile|syntax`	Pass
F12	`test_exit_status_dot_object_returns_zero_nonempty_stdout_empty_stderr`	0	empty	Pass

12/12 passed.

4.4 Split 4 Results

Frame	Test	Filter	Input	Result
F01	`test_select_numbers`	`select`	num_array	Pass
F02	`test_map_missing_keys`	`map`	obj_array (missing keys)	Pass
F03	`test_sort_numbers`	`sort`	num_array (unsorted)	Pass
F04	`test_group_by_objects`	`group_by`	obj_array (duplicates)	Pass
F05	`test_unique_mixed`	`unique`	mixed_array (duplicates)	Pass
F06	`test_add_numbers`	`add`	num_array	Pass
F07	`test_length_string`	`length`	string	Pass
F08	`test_select_missing_keys`	`select`	obj_array (missing keys)	Pass
F09	`test_map_mixed`	`map`	mixed_array	Pass
F10	`test_sort_empty`	`sort`	empty_array	Pass
F11	`test_group_by_numbers`	`group_by`	num_array (unsorted)	Pass
F12	`test_unique_objects`	`unique`	obj_array	Pass
F13	`test_add_empty`	`add`	empty_array	Pass
F14	`test_length_object`	`length`	object	Pass
F15	`test_length_null`	`length`	null	Pass

15/15 passed.

5. Overall Results

Split	Owner	Tests	Passed	Failed	Skipped
Split 1	Jeffrey Wong	18	18	0	0
Split 2	Tommy Duong	30	30	0	0
Split 3	Dean Zhou	12	12	0	0
Split 4	Ronney Lok	15	15	0	0
Total		75	75	0	0

Test environment (macOS final run):

- Auther: dean Zhou
- Reviewer: Tommy Duong
- OS: macOS 15.7.3 (Darwin 24.6.0, arm64)
- Python: 3.10.11 / pytest 9.0.2
- jq: jq-1.7.1-apple
- Date: 2026-02-23

6. Bugs Found

jq Bugs

No confirmed jq defects were found across all four splits. All tested behaviors (input modes, output formatting, error handling, and transformation filters) conformed to the jq manual specification.

Version-Specific Observations (Not Bugs)

Split	Observation
Split 2	On Windows, jq writes `CRLF` line endings; byte-level assertions account for platform
Split 3	jq-1.7.1-apple returns exit code `5` for malformed JSON (vs `4` in jq-1.6); oracles use `rc != 0` for portability
Split 4	`length(null)` returns `0` in jq ≥ 1.6; older builds may raise an error; test accepts both

7. Observations and Conclusions

Split Observations

Split 1 Input Modes

- File and stdin produce identical output for the same content, confirming input source transparency.
- `-R` reliably bypasses JSON parsing even for malformed input.
- `-R -s` correctly concatenates all lines (with newlines) into a single JSON string.
- `-n` ignores all input regardless of source or content.
- Empty file produces no output (exit 0) in default mode and `[]` with `-s`.

Split 2 Output Formatting

- All four formatting modes (`-c`, `-r`, `-j`, `--raw-output0`) behave exactly as documented.
- Exact byte-level assertions caught that `--raw-output0` uses NUL (`\x00`) not newline as the record terminator.
- `-j` suppresses inter-record newlines but preserves embedded newlines within strings.
- Unicode and backslash escaping rules differ correctly between JSON mode (default/compact) and raw mode.

Split 3 Error Handling & Exit Status

- Exit codes align with the jq manual: `2` for file errors, `3` for compile errors, `1` for `-e` with false/null output, `4` for `-e` with no output.
- Stderr patterns are consistent and parseable across jq versions (1.6, 1.7, 1.8).
- The `-e` flag only affects exit code, not stdout content.

Split 4 Transform Filters

- `group_by` and `unique` both sort before grouping/deduplicating, consistent with the spec.
- `group_by(.id)` correctly places `null`-keyed objects first (null sorts before numbers in jq).
- `add` on `[]` correctly returns `null`.
- `map` on an array of objects with missing keys correctly produces `null` for absent fields.
- All seven tested builtins behave robustly across numeric, object, mixed, empty, and edge-case inputs.

ACTS decision

We selected ACTS for combinatorial testing to avoid spending time implementing and maintaining a custom generator. ACTS is a mature, documented tool that produces pairwise frames from parameter and constraint files. Encoding our model as text files made the process reproducible and allowed us to focus on modelling the input space and oracles rather than focusing on generation logic. ACTS output CSV frames that we map to fixtures and tests, which simplified automation and made reruns or model extensions straightforward.

Zero-bugs finding interpretation

Our test campaign did not reveal any confirmed jq defects within the features and parameter combinations we exercised. This likely reflects jq's maturity and strong existing test coverage for the behaviors we targeted (input modes, output formatting, exit-status semantics, and core filters). However, absence of observed defects is not a proof of correctness: our study used pairwise coverage so faults may still exist in higher-order interactions, platform-specific builds, or under stress/fuzz conditions.

Next steps

To ensure that the jqlang will keep continuing to have a robust program the next steps that can be taken are:

- extend combinatorial depth for high-risk parameter (3-way+)
- add targeted fuzzing or randomized tests for parsing and filter expressions,
- run cross-version/platform regressions, and

8. Appendix: Use of LLMs

LLMs were used throughout this project to assist in generating automation scripts, tests, fixtures, and analysis.