bowtie

Release 2023.12.7

Julian Berman

Dec 10, 2023

CONTENTS

Bowtie is a *meta*-validator of the JSON Schema specification, by which we mean it coordinates executing *other* validator implementations, collecting and reporting on their results.

To do so it defines a simple input/output protocol (specified in this JSON Schema which validator implementations can implement, and it provides a CLI which can execute supported implementations.

It's called Bowtie because it fans in lots of JSON then fans out lots of results: >·<. Looks like a bowtie, no? Also because it's elegant – we hope.

CHAPTER

ONE

CLI

Bowtie is a versatile tool which you can use to investigate any or all of the implementations it supports. Below are a few sample command lines you might be interested in.

Running Commands Against All Implementations

Many of Bowtie's subcommands take a `-i / --implementation` option which specifies which implementations you wish to run against. In general, these same subcommands allow repeating this argument multiple times to run across multiple implementations. In many or even most cases, you may be interested in running against *all* implementations Bowtie supports. For somewhat boring reasons (partially having to do with the GitHub API) this "run against all implementations" functionality is slightly nontrivial to implement in a seamless way, though doing so is nevertheless tracked in this issue.

In the interim, it's often convenient to use a local checkout of Bowtie in order to list this information.

Specifically, all supported implementations live in the `implementations/` directory, and therefore you can construct a string of `-i` arguments using a small bit of shell vomit. If you have cloned Bowtie to `/path/to/bowtie` you should be able to use `$(ls /path/to/bowtie/implementations/ | sed 's/^| /-i /')` in any command to expand out to all implementations. See *below* for a full example.

1.1 Validating a Specific Instance Against One or More Implementations

The *bowtie validate* subcommand can be used to test arbitrary schemas and instances against any implementation Bowtie supports.

Given some collection of implementations to check – here perhaps two Javascript implementations – it takes a single schema and one or more instances to check against it:

```
$ bowtie validate -i js-ajv -i js-hyperjump <(printf '{"type": "integer"}') <(printf 37)
↪<(printf '"foo"')
```

Note that the schema and instance arguments are expected to be files, and that therefore the above makes use of normal shell process substitution to pass some examples on the command line.

Piping this output to *bowtie summary* is often the intended outcome (though not always, as you also may upload the output it gives to https://bowtie.report/ as a local report). For summarizing the results in the terminal however, the above command when summarized produces:

```
$ bowtie validate -i js-ajv -i js-hyperjump <(printf '{"type": "integer"}') <(printf 37)
↪<(printf '"foo"') | bowtie summary
2023-11-02 15:43.10 [debug    ] Will speak                    dialect=https://json-
↪schema.org/draft/2020-12/schema
2023-11-02 15:43.10 [info     ] Finished                      count=1
                                    Bowtie

 Schema

┌──────────────────────┐┌──────────┬──────────────────┬──────────────────────────┐
│ {                    ││ Instance │ ajv (javascript) │ hyperjump-jsv (javascript)
│   "type": "integer"  ├┼──────────┼──────────────────┼──────────────────────────┤
│ }                    ││ 37       │ valid            │ valid
│                      ││ "foo"    │ invalid          │ invalid
└──────────────────────┘└──────────┴──────────────────┴──────────────────────────┘
                                  2 tests ran
```

1.2 Running a Single Test Suite File

To run the draft 7 `type`-keyword tests on the Lua `jsonschema` implementation, run:

```
$ bowtie suite -i lua-jsonschema https://github.com/json-schema-org/JSON-Schema-Test-
↪Suite/blob/main/tests/draft7/type.json | bowtie summary --show failures
```

1.3 Running the Official Suite Across All Implementations

The following will run all Draft 7 tests from the official test suite (which it will automatically retrieve) across all implementations supporting Draft 7, and generate an HTML report named `bowtie-report.html` in the current directory:

```
$ bowtie suite $(ls /path/to/bowtie/implementations/ | sed 's/^| /-i /') https://github.
↪com/json-schema-org/JSON-Schema-Test-Suite/tree/main/tests/draft7 | bowtie summary --
↪show failures
```

1.4 Running Test Suite Tests From Local Checkouts

Providing a local path to the test suite can be used as well, which is useful if you have local changes:

```
$ bowtie suite $(ls /path/to/bowtie/implementations/ | sed 's/^| /-i /') ~/path/to/json-
↪schema-org/suite/tests/draft2020-12/ | bowtie summary --show failures
```

1.5 Checking An Implementation Functions On Basic Input

If you wish to verify that a particular implementation works on your machine (e.g. if you suspect a problem with the container image, or otherwise aren't seeing results), you can run *bowtie smoke*. E.g., to verify the Golang `jsonschema` implementation is functioning, you can run:

```
$ bowtie smoke -i go-jsonschema
```

1.5.1 Reference

bowtie

A meta-validator for the JSON Schema specifications.

Bowtie gives you access to JSON Schema across every programming language and implementation.

It lets you compare implementations to each other, or to known correct results from the JSON Schema test suite.

If you don't know where to begin, `bowtie validate` (for checking what any given implementations think of your schema) or `bowtie suite` (for running the official test suite against implementations) are likely good places to start.

Full documentation can also be found at https://docs.bowtie.report

```
bowtie [OPTIONS] COMMAND [ARGS]...
```

Options

--version

Show the version and exit.

badges

Generate Bowtie badges from a previous run.

```
bowtie badges [OPTIONS] OUTPUT
```

Options

--input <input>

Arguments

OUTPUT

Required argument

info

Retrieve a particular implementation (harness)'s metadata.

```
bowtie info [OPTIONS]
```

Options

-f, --format <format>

What format to use for the output

Default
pretty if stdout is a tty, otherwise JSON

Options
json | pretty

-i, --implementation <image_names>

Required A docker image which implements the bowtie IO protocol.

run

Run a sequence of cases provided on standard input.

```
bowtie run [OPTIONS] [INPUT]
```

Options

-i, --implementation <image_names>

Required A docker image which implements the bowtie IO protocol.

-D, --dialect <dialect>

A URI or shortname identifying the dialect of each test case.Shortnames include: ['2019', '2019-09', '201909', '2020', '2020-12', '202012', '3', '4', '6', '7', 'draft2019-09', 'draft201909', 'draft2020-12', 'draft202012', 'draft3', 'draft4', 'draft6', 'draft7'].

Default
draft2020-12

-k <filter>

Only run cases whose description match the given glob pattern.

-x, --fail-fast

Fail immediately after the first error or disagreement.

-S, --set-schema, --no-set-schema

Explicitly set $schema in all (non-boolean) case schemas sent to implementations. Note this of course means what is passed to implementations will differ from what is provided in the input.

Default
False

-T, --read-timeout <SECONDS>

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
2.0

-V, --validate-implementations

When speaking to implementations (provided via -i), validate the requests and responses sent to them under Bowtie's JSON Schema specification. Generally, this option protects against broken Bowtie implementations and can be left at its default (of off) unless you are developing a new implementation container.

Arguments

INPUT

Optional argument

smoke

Smoke test one or more implementations for basic correctness.

```
bowtie smoke [OPTIONS]
```

Options

-q, --quiet

Don't print any output, just exit with nonzero status on failure.

-f, --format <format>

What format to use for the output

Default
pretty if stdout is a tty, otherwise JSON

Options
json | pretty

-i, --implementation <image_names>

Required A docker image which implements the bowtie IO protocol.

suite

Run test cases from the official JSON Schema test suite.

Supports a number of possible inputs:

- file paths found on the local file system containing tests, e.g.:
  - {PATH}/tests/draft7 to run the draft 7 version's tests out of a local checkout of the test suite
  - {PATH}/tests/draft7/foo.json to run just one file from a checkout
- URLs to the test suite repository hosted on GitHub, e.g.:

  - `https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/main/tests/draft7/` to run a version directly from any branch which exists in GitHub
  - `https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/main/tests/draft7/foo.json` to run a single file directly from a branch which exists in GitHub
- short name versions of the previous URLs (similar to those providable to `bowtie validate` via its `--dialect` option), e.g.:
  - `7`, to run the draft 7 tests directly from GitHub (as in the URL example above)

```
bowtie suite [OPTIONS] INPUT
```

Options

-i, --implementation <image_names>

Required A docker image which implements the bowtie IO protocol.

-k <filter>

Only run cases whose description match the given glob pattern.

-x, --fail-fast

Fail immediately after the first error or disagreement.

-S, --set-schema, --no-set-schema

Explicitly set $schema in all (non-boolean) case schemas sent to implementations. Note this of course means what is passed to implementations will differ from what is provided in the input.

Default
`False`

-T, --read-timeout <SECONDS>

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
`2.0`

-V, --validate-implementations

When speaking to implementations (provided via -i), validate the requests and responses sent to them under Bowtie's JSON Schema specification. Generally, this option protects against broken Bowtie implementations and can be left at its default (of off) unless you are developing a new implementation container.

Arguments

INPUT

Required argument

summary

Generate an (in-terminal) summary of a Bowtie run.

```
bowtie summary [OPTIONS] [INPUT]
```

Options

-f, --format <format>

What format to use for the output

Default
`pretty if stdout is a tty, otherwise JSON`

Options
json | pretty

-s, --show <show>

Configure whether to display validation results (whether instances are valid or not) or test failure results (whether the validation results match expected validation results)

Default
`validation`

Options
failures | validation

Arguments

INPUT

Optional argument

tui

Open a simple interactive TUI for executing Bowtie commands.

```
bowtie tui [OPTIONS]
```

validate

Validate one or more instances under a given schema across implementations.

```
bowtie validate [OPTIONS] SCHEMA [INSTANCES]...
```

Options

-i, --implementation <image_names>

Required A docker image which implements the bowtie IO protocol.

-D, --dialect <dialect>

A URI or shortname identifying the dialect of each test case.Shortnames include: [‘2019’, ‘2019-09’, ‘201909’, ‘2020’, ‘2020-12’, ‘202012’, ‘3’, ‘4’, ‘6’, ‘7’, ‘draft2019-09’, ‘draft201909’, ‘draft2020-12’, ‘draft202012’, ‘draft3’, ‘draft4’, ‘draft6’, ‘draft7’].

Default
draft2020-12

-S, --set-schema, --no-set-schema

Explicitly set $schema in all (non-boolean) case schemas sent to implementations. Note this of course means what is passed to implementations will differ from what is provided in the input.

Default
False

-T, --read-timeout <SECONDS>

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
2.0

-V, --validate-implementations

When speaking to implementations (provided via -i), validate the requests and responses sent to them under Bowtie’s JSON Schema specification. Generally, this option protects against broken Bowtie implementations and can be left at its default (of off) unless you are developing a new implementation container.

--expect <expect>

Expect the given input to be considered valid or invalid, or else (with ‘any’) to allow either result.

Default
any

Options
valid | invalid | any

Arguments

SCHEMA

Required argument

INSTANCES

Optional argument(s)

CHAPTER

TWO

TUTORIAL: ADDING AN IMPLEMENTATION

The purpose of *Bowtie* is to support passing instances and schemas through a large number of JSON Schema implementations, collecting their output for comparison.

If you've written or used an implementation of JSON Schema which isn't already supported, let's see how you can add support for it to Bowtie.

Bowtie orchestrates running a number of containers, passing JSON Schema test cases to each one of them, and then collecting and comparing results across implementations. Any JSON Schema implementation will have some way of getting schemas and instances "into" it for processing. We'll wrap this implementation-specific API inside a small harness which accepts input from Bowtie over standard input and writes results to standard output in the format Bowtie expects.

As a last step before we get into details, let's summarize some terminology (which you can also skip and refer back to if needed):

implementation
: a library or program which implements the JSON Schema specification

validation API
: the specific function(s), method(s), or similar constructs within the *implementation* which cause it to evaluate a schema against a specific instance

host language
: the programming language which a particular *implementation* is written in

test harness
test runner
: a small program which accepts *test cases* sent from Bowtie, passes them through a specific *implementation's validation API*, and writes the results of this validation process out for Bowtie to read

harness language
: the programming language which the *test harness* itself is written in. Typically this will match the *host language*, since doing so will make it easier to call out directly to the *implementation*.

test case
: a specific JSON schema and instance which Bowtie will pass to any *implementations* it is testing

IHOP
: the *i*nput → *h*arness → *o*utput *p*rotocol. A JSON protocol governing the structure and semantics of messages which Bowtie will send to *test harnesses* as well as the structure and semantics it expects from JSON responses sent back.

2.1 Prerequisites

- Bowtie itself, already installed on your machine
- A target *implementation*, which you do *not* necessarily need installed on your machine
- docker, podman or a similarly compatible tool for building OCI container images and running OCI containers

2.2 Step 0: Familiarization With the Implementation

Once you've installed the prerequisites, your first step is to ensure you're familiar with the implementation you're about to add support for, as well as with its *host language*. If you're its author, you're certainly well qualified :) – if not, you'll see a few things below which you'll need to find in its API documentation, such as what function(s) or object(s) you use to validate instances under schemas. If you're not, there shouldn't be a need to be an expert neither in the language nor implementation, as we'll be writing only a small wrapper program, but you definitely will need to know how to compile or run programs in the host language, how to read and write JSON from it, and how to package programs into container images.

For the purposes of this tutorial, we'll write support for a Lua implementation of JSON Schema (one which calls itself simply `jsonschema` within the Lua ecosystem, as many implementations tend to). Bowtie of course already supports this implementation officially, so if you want to see the final result either now or at the end of this tutorial, it's here. If you're not already familiar with Lua as a programming language, the below won't serve as a full tutorial of course, but you still should be able to follow along; it's a fairly simple one.

Let's get a Hello World container running which we'll turn into our test harness.

Create a directory somewhere, and within it create a `Dockerfile` with these contents:

```
FROM alpine:3.16
RUN apk add --no-cache luajit luajit-dev pcre-dev gcc libc-dev curl make cmake && \
    wget 'https://luarocks.org/releases/luarocks-3.9.1.tar.gz' && \
    tar -xf luarocks-3.9.1.tar.gz && cd luarocks-3.9.1 && \
    ./configure && make && make install
RUN sed -i '/WGET/d' /usr/local/share/lua/5.1/luarocks/fs/tools.lua
RUN luarocks install jsonschema
ADD https://raw.githubusercontent.com/rxi/json.lua/master/json.lua .
COPY bowtie_jsonschema.lua .
CMD ["luajit", "bowtie_jsonschema.lua"]
```

Most of the above is slightly *more* complicated than one you'll need to write for your own language, and has to do with some Lua packaging issues that are uninteresting to discuss in detail. The notable bit is we'll create a `bowtie_jsonschema.lua` file, our test harness, and then this container image will invoke it. Bowtie will then speak to the running harness container.

Let's check everything works. Create a file named `bowtie_jsonschema.lua` with these contents:

```
print('Hello world')
```

and build the image (below using `podman` but if you're using `docker`, just substitute it for `podman` in all commands below):

```
podman build --quiet -f Dockerfile -t bowtie-lua-jsonschema .
```

Note: If you are indeed using `podman`, you must ensure you have set the `DOCKER_HOST` environment variable in the environment in which you invoke `bowtie`.

This ensures `bowtie` can speak the Docker API to your `podman` installation (which is needed because the API client used within Bowtie is agnostic, but speaks the Docker API, which `podman` supports as well).

Further information may be found here.

If everything went well, running:

```
podman run --rm bowtie-lua-jsonschema
```

should get you some output:

```
Hello world
```

We're off to the races.

2.3 Step 1: IHOP Here We Come

From here on out we'll continue to modify the `bowtie_jsonschema.lua` file we created above, so keep that file open and make changes to it as directed below.

Bowtie sends JSON requests (or commands) to *test harness* containers over standard input, and expects responses to be sent over standard output.

Each request has a `cmd` property which specifies which type of request it is. Additional properties are arguments to the request.

There are 4 commands every harness must know how to respond to, shown here as a brief excerpt from the full schema specifying the protocol:

```
"oneOf": [
  { "$ref": "tag:bowtie.report,2023:ihop:command:start" },
  { "$ref": "tag:bowtie.report,2023:ihop:command:dialect" },
  { "$ref": "tag:bowtie.report,2023:ihop:command:run" },
  { "$ref": "tag:bowtie.report,2023:ihop:command:stop" }
]
```

Bowtie will send a `start` request when first starting a harness, and then at the end will send a `stop` request telling the harness to shut down.

Let's start filling out a real test harness implementation by at least reacting to `start` and `stop` requests.

Note: From here on out in this document it's assumed you rebuild your container image each time you modify the harness via

```
podman build -f Dockerfile -t localhost/tutorial-lua-jsonschema .
```

Change your harness to contain:

```
local json = require 'json'
```

(continues on next page)

(continued from previous page)

```
io.stdout:setvbuf 'line'

local cmds = {
  start = function(_)
    io.stderr:write("STARTING")
    return {}
  end,

  stop = function(_)
    io.stderr:write("STOPPING")
    return {}
  end,
}

for line in io.lines() do
  local request = json.decode(line)
  local response = cmds[request.cmd](request)
  io.write(json.encode(response) .. '\n')
end
```

If this is your first time reading Lua code, what we've done is create a dispatch table (a mapping from string command names to functions handling each one), and implemented stub functions which simply write to `stderr` when called, and then return empty responses.

Note: We also have configured `stdout` for the harness to be line-buffered (by calling `setvbuf`). Ensure you've done the equivalent for your host language, as Bowtie expects to be able to read responses to each message it sends even though some languages do not necessarily flush their output on each line write.

Any other command other than `start` or `stop` will blow up, but we're reading and writing JSON!

Let's see what happens if we use this nonetheless. We can pass a hand-crafted *test case* to Bowtie by running:

```
bowtie run -i localhost/tutorial-lua-jsonschema <<EOF
    {"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪"instance": {}}] }
    {"description": "test case 2", "schema": {"const": 37}, "tests": [{"description":
↪"not 37", "instance": {}}, {"description": "is 37", "instance": 37}] }
EOF
```

which if you now run should produce something like:

```
2022-10-05 15:39.59 [debug    ] Will speak dialect            dialect=https://json-
↪schema.org/draft/2020-12/schema
Traceback (most recent call last):
    ...
TypeError: bowtie._commands.Started() argument after ** must be a mapping, not list
```

… a nice big inscrutable error message. Our harness isn't returning valid responses, and Bowtie doesn't know how to handle what we've sent it. The structure of the protocol we're trying to implement lives in a JSON Schema though, and Bowtie can be told to validate requests and responses using the schema. You can enable this validation by passing **`bowtie run`** the *`-V`* option, which will produce nicer messages while we develop the harness:

```
bowtie run -i localhost/tutorial-lua-jsonschema -V <<EOF
{"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪"instance": {}}] }
{"description": "test case 2", "schema": {"const": 37}, "tests": [{"description": "not 37
↪", "instance": {}}, {"description": "is 37", "instance": 37}] }
EOF
2022-10-05 20:59.41 [debug    ] Will speak dialect             dialect=https://json-
↪schema.org/draft/2020-12/schema
2022-10-05 20:59.41 [error    ] Invalid response               [localhost/tutorial-lua-
↪jsonschema] errors=[<ValidationError: "[] is not of type 'object'">]␣
↪request=Start(version=1)
2022-10-05 20:59.45 [warning  ] Unsupported dialect, skipping implementation. [localhost/
↪tutorial-lua-jsonschema] dialect=https://json-schema.org/draft/2020-12/schema
{"implementations": {}}
{"case": {"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪ "instance": {}, "valid": null}], "comment": null, "registry": null}, "seq": 1}
{"case": {"description": "test case 2", "schema": {"const": 37}, "tests": [{"description
↪": "not 37", "instance": {}, "valid": null}, {"description": "is 37", "instance": 37,
↪"valid": null}], "comment": null, "registry": null}, "seq": 2}
2022-10-05 20:59.45 [info     ] Finished                       count=2
```

which is telling us we're returning JSON arrays to Bowtie instead of JSON objects. Lua the language has only one container type (`table`), and we've returned `{}` which the JSON library guesses means "empty array". Don't think too hard about Lua's peculiarities, let's just fix it by having a look at what parameters the `start` command sends a harness and what it expects back. The schema says:

```
{
  "description": "Sent once at program start to the implementation to indicate Bowtie is␣
↪starting to send test cases.",

  "$id": "tag:bowtie.report,2023:ihop:command:start",

  "required": ["version"],
  "properties": {
    "cmd": { "const": "start" },
    "version": {
      "description": "The version of the Bowtie protocol which is intended.",
      "$ref": "tag:bowtie.report,2023:ihop#version"
    }
  },
```

so `start` requests will have two parameters:

- `cmd` which indicates the kind of request being sent (and is present in all requests sent by Bowtie)
- `version` which represents the version of the Bowtie protocol being spoken. Today, that version is always `1`, but that may change in the future, in which case a harness should bail out as it may not understand the requests being sent.

The harness is expected to respond with something conforming to:

```
    "response": {
      "$anchor": "response",

      "type": "object",
```

(continues on next page)

(continued from previous page)

```
      "required": ["ready", "version", "implementation"],
      "properties": {
        "ready": {
          "description": "Confirmation that the implementation is ready.",
          "const": true
        },
        "version": {
          "description": "Confirmation of the Bowtie version",
          "$ref": "tag:bowtie.report,2023:ihop#version"
        },
        "implementation": {
          "description": "Metadata about the implementation. The list below contain␣
↪required or suggested values, but implementation-specific additional metadata can also␣
↪be included and will be preserved in emitted reports.",

          "type": "object",
          "required": ["name", "language", "dialects", "homepage", "issues"],
          "properties": {
            "language": {
              "description": "The implementation language (e.g. C++, Python, etc.)",

              "type": "string",
              "pattern": "^[a-z0-9-+_]*$"
            },
            "name": {
              "description": "The name of the implementation itself",

              "type": "string",
              "pattern": "^[A-Za-z][\\w\\-.]*[A-Za-z0-9]$"
            },
            "version": {
              "description": "The implementation version",
              "type": "string"
            },
            "dialects": {
              "description": "A list of JSON Schema dialects (URIs) which the␣
↪implementation understands. When running test cases, this list will be consulted␣
↪before sending them to the implementation (and any unsupported dialects will be␣
↪skipped).",

              "type": "array",
              "items": { "type": "string", "format": "uri" }
            },
            "homepage": {
              "description": "A URL for the implementation's homepage",

              "type": "string",
              "format": "uri"
            },
            "issues": {
              "description": "A URL for the implementation's bug tracker",
```

(continues on next page)

(continued from previous page)

```
            "type": "string",
            "format": "uri"
          },
          "links": {
            "description": "Additional web page links relevant to the implementation",

            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "description": { "type": "string" },
                "url": { "type": "string", "format": "uri" }
              }
            }
          },
          "os": {
            "description": "Operating system the implementation is running on",
            "type": "string"
          },
          "os_version": {
            "description": "Version of OS the implementation is running on",
            "type": "string"
          },
          "language_version": {
            "description": "Version of language used to run the implementation",
            "type": "string"
          }
        }
      }
    }
  }
}
```

which is some metadata about the implementation being tested, and includes things like:

- its name
- a URL for its bug tracker for use if issues are found
- the versions of JSON Schema it supports, identified via URIs

You can also have a look at the full schema for details on the `stop` command, but it essentially doesn't require a response, and simply signals the harness that it should exit.

Let's implement both requests. Change your harness to contain:

```
local json = require 'json'

STARTED = false

local cmds = {
  start = function(request)
    assert(request.version == 1, 'Wrong version!')
    STARTED = true
```

(continues on next page)

(continued from previous page)

```
    return {
      ready = true,
      version = 1,
      implementation = {
        language = 'lua',
        name = 'jsonschema',
        homepage = 'https://github.com/api7/jsonschema',
        issues = 'https://github.com/api7/jsonschema/issues',

        dialects = {
          'http://json-schema.org/draft-07/schema#',
          'http://json-schema.org/draft-06/schema#',
          'http://json-schema.org/draft-04/schema#',
        },
      },
    }
  end,

  stop = function(_)
    assert(STARTED, 'Not started!')
    os.exit(0)
  end,
}

for line in io.lines() do
  local request = json.decode(line)
  local response = cmds[request.cmd](request)
  io.write(json.encode(response) .. '\n')
end
```

We now return some detail about the implementation when responding to `start` requests (sent as JSON), including which versions of the specification it supports.

When stopping, we simply exit successfully.

If you re-run `bowtie`, you'll see now that it doesn't crash, though it outputs:

```
2022-10-11 13:44.40 [debug    ] Will speak dialect             dialect=https://json-
↪schema.org/draft/2020-12/schema
2022-10-11 13:44.40 [warning  ] Unsupported dialect, skipping implementation. [localhost/
↪tutorial-lua-jsonschema] dialect=https://json-schema.org/draft/2020-12/schema
{"implementations": {}}
```

Our harness is now properly starting and stopping, but this Lua implementation only supports versions of JSON Schema earlier than Draft 7, and Bowtie is defaulting to a newer version. Tell Bowtie we are speaking an earlier version by passing the *`--dialect`* option, i.e. `bowtie run --dialect 7 -i localhost/tutorial-lua-jsonschema`.

Tip: Any of `7`, `draft7`, or the full draft 7 meta schema URI will work to set the dialect in use.

If we yet again invoke `bowtie`, we now see something like:

```
2022-10-31 12:26.05 [debug    ] Will speak dialect            dialect=http://json-
↪schema.org/draft-07/schema#
──────────── localhost/tutorial-lua-jsonschema (stderr) ────────────
│                                                                  │
│    luajit: bowtie_jsonschema.lua:35: attempt to call a nil value │
│    stack traceback:                                              │
│            bowtie_jsonschema.lua:35: in main chunk               │
│            [C]: at 0xaaaad55c8690                                │
│                                                                  │
──────────────────────────────────────────────────────────────────
2022-10-31 12:26.06 [error    ] Tried to start sending test cases, but got an error.␣
↪[localhost/tutorial-lua-jsonschema]
2022-10-31 12:26.06 [error    ] No implementations started successfully!
```

which is indicating that we have yet another command to implement – the `dialect` command.

2.4 Step 2: Configuring Implicit Dialects

The JSON Schema specification generally allows schemas of the form `{"type": "object"}` – i.e. ones where the schema does not internally include a $schema keyword which would otherwise indicate the dialect of JSON Schema being used. In other words, the aforementioned schema may be treated (depending on its author's intention) as a Draft 4 schema, a Draft 7 schema, a Draft 2020-12 schema, etc. Bowtie enables specifying an intended behavior for such schemas by communicating it "out-of-band" to harnesses via the `dialect` command, which indicates to the harness: "treat schemas without `$schema` as this particular dialect (provided in the request)". The structure of this command looks like:

```
{
  "description": "Sent to indicate an 'implicit dialect' -- i.e. which dialect is␣
↪intended for schemas which do not contain $schema. May be sent multiple times by␣
↪bowtie to indicate a change in implicit dialect. Note that implementations do not have␣
↪to support processing such schemas in certain versions of the JSON Schema␣
↪specification. Harnesses should *not* attempt to pepper this behavior over, or␣
↪otherwise change the behavior of an implementation. In other words, this value should␣
↪*not* be used to mutate incoming schemas (by inserting $schema). If an implementation␣
↪does not support indicating what dialect a schema is written against unless indicated␣
↪by $schema, or if it refuses to process schemas which do not contain $schema entirely,␣
↪it should respond to this command as indicated below, and simply error when running␣
↪cases containing such schemas as it would when used normally. Regardless of the␣
↪dialect specified by this request, *explicitly* dialected schemas (which do contain
↪$schema) with a different dialect may still be sent (as long as the implementation has␣
↪signalled it supports the dialect)!",

  "$id": "tag:bowtie.report,2023:ihop:command:dialect",

  "properties": {
    "cmd": { "const": "dialect" },
    "dialect": {
      "description": "A dialect URI which has previously been recognized as supported by␣
↪the implementation's start response.",

      "type": "string",
```

(continues on next page)

(continued from previous page)

```
      "format": "uri"
    }
  },
```

We need the harness to accept the incoming dialect request and configure the Lua implementation that we're wrapping to treat `$schema`-less schemas as the specified dialect – only there's a catch. This implementation actually doesn't support a mechanism for specifying how to treat these kinds of schemas.

See also:

The harness for the Rust jsonschema-rs implementation
for an example of an implementation which does support self-configuration for the `dialect` command.

In this case, the harness should flag this to Bowtie so that it is aware that the harness is unable to configure the implementation in this way, which we do by simply responding with `{"ok": false}`. Add a handler for the `dialect` command to your harness which returns that response:

```
dialect = function(_)
  assert(STARTED, 'Not started!')
  return { ok = false }
end,
```

Warning: Responding `{"ok": true}` or `false` is *not* an indication of whether an implementation supports the dialect sent.

Bowtie will never send a dialect request for a dialect that a harness does not support – information which it already knows from the `start` response we implemented *earlier*, where we specified the complete list of supported dialects for the implementation. If it ever did so this would be considered a Bowtie bug.

This `dialect` request *strictly* controls setting what an implementation harness should do with schemas that do *not* internally indicate what version they are written for; its response should signal only whether the implementation has configured itself appropriately or whether doing so is not supported.

Bowtie will *continue executing tests* even if it sees a `false` response.

Running `bowtie` now should produce:

```
2022-10-31 13:04.51 [debug    ] Will speak dialect            dialect=http://json-
↪schema.org/draft-07/schema#
2022-10-31 13:04.52 [warning  ] Implicit dialect not acknowledged. Proceeding, but␣
↪implementation may not have configured itself to handle schemas without $schema.␣
↪[localhost/tutorial-lua-jsonschema] dialect=http://json-schema.org/draft-07/schema#␣
↪response=StartedDialect(ok=False)
{"implementations": {"localhost/tutorial-lua-jsonschema": {"language": "lua", "name":
↪"jsonschema", "homepage": "https://github.com/api7/jsonschema", "issues": "https://
↪github.com/api7/jsonschema/issues", "dialects": ["http://json-schema.org/draft-07/
↪schema#", "http://json-schema.org/draft-06/schema#", "http://json-schema.org/draft-04/
↪schema#"], "image": "localhost/tutorial-lua-jsonschema"}}}
{"case": {"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪ "instance": {}, "valid": null}], "comment": null, "registry": null}, "seq": 1}
──────────── localhost/tutorial-lua-jsonschema (stderr) ────────────
┌──────────────────────────────────────────────────────────────────┐
│    luajit: bowtie_jsonschema.lua:42: attempt to call a nil value │
│    stack traceback:                                              │
```

(continues on next page)

(continued from previous page)

```
│            bowtie_jsonschema.lua:42: in main chunk                    │
│            [C]: at 0xaaaacecf8690                                     │
│                                                                       │
──────────────────────────────────────────────────────────────────────────
2022-10-31 13:04.52 [error    ] uncaught error                [localhost/tutorial-lua-
↪jsonschema] case=test case 1 schema={} seq=1
{"implementation": "localhost/tutorial-lua-jsonschema", "seq": 1, "context": {"stderr":
↪"luajit: bowtie_jsonschema.lua:42: attempt to call a nil value\nstack traceback:\n\
↪tbowtie_jsonschema.lua:42: in main chunk\n\t[C]: at 0xaaaacecf8690\n"}, "caught":␣
↪false}
{"case": {"description": "test case 2", "schema": {"const": 37}, "tests": [{"description
↪": "not 37", "instance": {}, "valid": null}, {"description": "is 37", "instance": 37,
↪"valid": null}], "comment": null, "registry": null}, "seq": 2}
──────────── localhost/tutorial-lua-jsonschema (stderr) ────────────
│                                                                       │
│    luajit: bowtie_jsonschema.lua:42: attempt to call a nil value      │
│    stack traceback:                                                   │
│            bowtie_jsonschema.lua:42: in main chunk                    │
│            [C]: at 0xaaaae3078690                                     │
│                                                                       │
──────────────────────────────────────────────────────────────────────────
2022-10-31 13:04.53 [error    ] uncaught error                [localhost/tutorial-lua-
↪jsonschema] case=test case 2 schema={'const': 37} seq=2
{"implementation": "localhost/tutorial-lua-jsonschema", "seq": 2, "context": {"stderr":
↪"luajit: bowtie_jsonschema.lua:42: attempt to call a nil value\nstack traceback:\n\
↪tbowtie_jsonschema.lua:42: in main chunk\n\t[C]: at 0xaaaae3078690\n"}, "caught":␣
↪false}
2022-10-31 13:04.53 [info     ] Finished                      count=2
```

where we have one final command left to implement – actually running our test cases!

2.5 Step 3: Validating Instances

It's now time to actually invoke the implementation itself, so import the library by adding:

```
local jsonschema = require 'jsonschema'
```

to the top of your harness.

The API we need within the Lua `jsonschema` library is one called `jsonschema.generate_validator`, which is the API which a user of the library calls in order to validate an instance under a provided schema. For details on how to use this API, see the library's documentation, but for our purposes it essentially takes 2 arguments – a JSON Schema (represented as a Lua `table`) along with an additional options argument which we'll use momentarily. It returns a callable which then can be used to validate instances (other Lua values). Let's take a first pass at implementing the `run` command, whose input looks like:

```
{
  "description": "Sent to implementations for each test case.",

  "$id": "tag:bowtie.report,2023:ihop:command:run",
```

(continues on next page)

(continued from previous page)

```
  "required": ["seq", "case"],
  "properties": {
    "cmd": { "const": "run" },
    "seq": {
      "description": "A sequence identifier for the test case. It must be passed along␣
↪as-is in the response.",
      "$anchor": "seq"
    },
    "case": { "$ref": "tag:bowtie.report,2023:ihop#case" }
  },
```

`run` requests contain a test case (a schema with tests), alongside a `seq` parameter which is simply an identifier for the request and needs to be included in the response we write back. Here's an implementation of the `run` command to add to our harness implementation:

```
run = function(request)
  assert(STARTED, 'Not started!')

  local validate = jsonschema.generate_validator(request.case.schema)
  local results = {}
  for _, test in ipairs(request.case.tests) do
    table.insert(results, { valid = validate(test.instance) })
  end
  return { seq = request.seq, results = results }
end,
```

We call `generate_validator` to get our validation callable, then we apply it (`map` it, though Lua has no builtin to do so) over all tests in the `run` request, returning a response which contains the `seq` number alongside results for each test. The results are indicated positionally as shown above, meaning the first result in the results array should be the result for the first test in the input array.

If we run `bowtie` again, we see:

```
2022-10-31 13:20.14 [debug    ] Will speak dialect            dialect=http://json-
↪schema.org/draft-07/schema#
2022-10-31 13:20.14 [warning  ] Implicit dialect not acknowledged. Proceeding, but␣
↪implementation may not have configured itself to handle schemas without $schema.␣
↪[localhost/tutorial-lua-jsonschema] dialect=http://json-schema.org/draft-07/schema#␣
↪response=StartedDialect(ok=False)
{"implementations": {"localhost/tutorial-lua-jsonschema": {"dialects": ["http://json-
↪schema.org/draft-07/schema#", "http://json-schema.org/draft-06/schema#", "http://json-
↪schema.org/draft-04/schema#"], "language": "lua", "name": "jsonschema", "homepage":
↪"https://github.com/api7/jsonschema", "issues": "https://github.com/api7/jsonschema/
↪issues", "image": "localhost/tutorial-lua-jsonschema"}}}
{"case": {"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪ "instance": {}, "valid": null}], "comment": null, "registry": null}, "seq": 1}
{"implementation": "localhost/tutorial-lua-jsonschema", "seq": 1, "results": [{"valid":␣
↪true}], "expected": [null]}
{"case": {"description": "test case 2", "schema": {"const": 37}, "tests": [{"description
↪": "not 37", "instance": {}, "valid": null}, {"description": "is 37", "instance": 37,
↪"valid": null}], "comment": null, "registry": null}, "seq": 2}
{"implementation": "localhost/tutorial-lua-jsonschema", "seq": 2, "results": [{"valid":␣
↪false}, {"valid": true}], "expected": [null, null]}
```

(continues on next page)

(continued from previous page)

```
2022-10-31 13:20.14 [info     ] Finished                       count=2
```

where we've now successfully run some inputted test cases. The output we see now contains the results returned by the Lua implementation and is ready to be piped into *bowtie summary*. Hooray!

2.6 Step 4: Resolving References

In order to support testing the $ref keyword from JSON Schema, which involves resolving references to JSON documents, there's an additional parameter that is sent with `run` commands which contains a schema *registry*, i.e. a collection of additional schemas beyond the test case schema itself which may be referenced from within the test case. The intention is that the harness should configure its implementation to be able to retrieve any of the schemas present in the registry for the duration of the test case.

As an example, a registry may look like:

```
{
  "http://example.com/my/string/schema": {"type": "string"}
}
```

which, if included in a `run` request means that the implementation is expected to resolve a retrieval URI of `http://example.com/my/string/schema` to the corresponding schema above. Each key in the registry is an (absolute) retrieval URI and each value is a corresponding JSON Schema.

The Lua implementation we have been writing a harness for actually contains no built-in support for resolving `$ref` by default, but does give us a way to "hook in" an implementation of reference resolution. Specifically, `generate_validator` takes a second argument (a Lua `table`) whose `external_resolver` key may point to a callable that is called when encountering a reference. If we pass the library a callable which simply retrieves references from the registry provided whenever encountering a `$ref`, we have implemented what's needed by Bowtie. Change the call to `generate_validator` to look like:

```
local validate = jsonschema.generate_validator(request.case.schema, {
  external_resolver = function(url)
    return request.case.registry[url]
  end,
})
```

where we simply index into `request.case.registry` anytime we see a referenced URL.

And *now* it would seem we're done. We could stop here, as we've done enough that Bowtie can take it from here, but there are a few finishing touches to implement which improve performance or overall experience interacting with our new harness, so let's get to those before we clean up shop.

2.7 Step 5: Handling Errors

If an implementation happens to not be fully compliant with the specification, or not fully compliant *yet*, Bowtie may end up passing a schema or instance that causes the harness to crash if the underlying implementation crashes (by panicking, raising an exception, etc. depending on the host language).

Whilst Bowtie tries to be hearty about these possibilities by automatically restarting crashed containers, it's more efficient for the harness itself to do so via mechanisms within the host language.

Bowtie will show an `uncaught error` message in its debugging output whenever a container crashes. We can make the harness internally catch the error(s) and return a special response to `run` requests which signals that the implementation errored. Doing so will still be marked as an error in debugging output, but Bowtie will recognize that the error was caught by the harness, and things will generally be faster by not incurring additional restart cost each time the harness crashes.

Catching exceptions from our Lua implementation is simple, by wrapping the `validate()` function call with the `pcall` function, which will catch any errors raised and allow us to detect whether any have occurred. Once the harness detects an error, it should return an error response (in place of results), which may include any diagnostic information for later use, e.g. a traceback or internal error message. The structure of error responses is:

```
        "errored": {
          "description": "Signal that the implementation encountered an internal error␣
↪(which it caught). Additional context can be passed along (e.g. a traceback or␣
↪exception detail).",

          "$anchor": "errored",

          "required": ["errored"],
          "properties": {
            "errored": { "const": true },
            "context": {
              "description": "Additional implementation-specific or language-specific␣
↪context available when the error was caught.",
              "type": "object",
              "properties": {
                "message": {
                  "description": "A (short) description of the error.",
                  "type": "string"
                },
                "traceback": {
                  "description": "A language-specific traceback (backtrace) containing␣
↪stack frames or fuller debugging information on where the error occurred.",
                  "type": "string"
                },
                "stderr": {
                  "description": "The raw (captured) contents of standard error from␣
↪within the harness. Prefer using 'traceback' if the contents are a traceback.",
                  "type": "string"
                }
              },
              "additionalProperties": true
            }
          }
        }
      }
    }
  }
}
```

(i.e. in particular the harness should return a response setting `errored` to `true`).

2.8 Step 6: Skipping Tests & Handling Known Issues

The aforementioned error-handling support means that a running harness can be fairly sure it gracefully continues in the face of issues, even for unexpectedly new input.

But if an implementation has *known* gaps, it's better to explicitly tell Bowtie that a particular feature is unsupported, rather than simply letting the error handling code trap any exception or error. The reasoning here is that Bowtie can surface any detail about *why* a test is skipped, or some day may even notice when an issue on an implementation is closed (and retry running the test). In general, it is preferable that a test harness have 0 errors, and certainly 0 uncaught errors, when being run under the official suite (and instead intentionally skip known issues, while emitting a link to an issue tracking the missing functionality). Support for skipping tests is still somewhat crude, but it does indeed work, and should be preferred by an implementer if you know a particular test from the official suite isn't supported by your implementation.

The structure of skip responses, which you should send when presented with a "known" unsupported test, is:

```
"skipped": {
  "description": "Signal that the implementation skipped the test case or test␣
↪(typically because it is a known bug). Either an issue URL or a human-readable message␣
↪is encouraged to explain the skip.",

  "$anchor": "skipped",

  "required": ["skipped"],
  "properties": {
    "skipped": { "const": true },
    "message": {
      "description": "A human-readable message passed back from the implementation␣
↪explaining why the skip occurred.",
      "type": "string"
    },
    "issue_url": {
      "description": "An optional link to a relevant issue on the implementation's bug␣
↪tracker.",

      "type": "string",
      "format": "url"
    }
  }
},
```

The biggest consideration at the minute is how to *identify* incoming test cases. Adding some sort of "persistent identifier" to test cases is something we've previously discussed upstream in the official test suite, which would make this easier. Until that happens however, your best current bet is to match on the test case description and/or schema, and use that to decide this incoming test is unsupported (and then respond with a skip request as above). For a specific example of doing so for Bowtie's reporting, see this PR.

If you've gotten to the end and wish to see the full code for the harness, have a look at the completed harness for lua-jsonschema.

2.9 Addendum: Submitting Upstream

If the implementation you've added isn't already supported by Bowtie, contributing it is very welcome!

Please feel free to open an issue or pull request adding it to the implementations directory.

If you do so there are a few additional things to do beyond the above:

- Commit your harness to the `implementations` directory in the root of Bowtie's repository, alongside the existing ones. Name your harness directory `<host-language>-<implementation-name>/`. Use ASCII-compatible names, so if your implementation is written in C++ and is called `flooblekins` within the C++ ecosystem, call the directory `cpp-flooblekins/`.
- Please ensure you've used an `alpine`-based image, or at least a slim one, to keep sizes as small as possible. Reference the existing `Dockerfile`s if you need inspiration.
- Please also add some sort of linter or autoformatter for the source code you've written. Because Bowtie has code from so many languages in it, it's simply too much to expect any human eyes to catch style or formatting issues. Which autoformatter you use will depend on the host language, but if you look at the .pre-commit-config.yaml file in this repository you'll see we run `go fmt` for Golang implementations, `cargo fmt` for Rust ones, `black` for Python ones, etc., each being a "commonly used" tool in the corresponding language. If yours is the first implementation Bowtie supports in a language, use the "most commonly used" linter or autoformatter for the language.
- Finally, please enable dependabot support for your implementation whenever possible (or necessary). If your image does not pin the library being tested this may not be necessary (since each rebuild of the image should retrieve the latest version), but if it does (via e.g. a `package.json` in Javascript, a `requirements.txt` in Python, a `cargo.lock` for Rust, etc.) Dependabot can be used to automatically ensure we update Bowtie's implementation versions without manual labor.

CHAPTER

THREE

CONTRIBUTOR'S GUIDE

Thanks for considering contributing to Bowtie, it'd be very much appreciated!

3.1 Installation

If you're going to work on Bowtie itself, you likely will want to install it using Python's editable install functionality, e.g.:

```
$ pip install -r bowtie-repo/requirements.txt -e bowtie-repo/
```

which will allow you to make changes to files within Bowtie and see the results without reinstalling it repeatedly.

3.2 Running the Tests

Bowtie has a small set of integration tests which ensure it works correctly on a number of cases.

You can find them in the `tests/` directory.

You can run them using nox, which you can install using the linked instructions, and then can run:

```
$ nox -s tests-3.11
```

to run the tests using Python 3.11 (or any other version you'd like).

Before submitting a PR you may want to run the full suite of tests by running `nox` with no arguments to run all environments.

3.3 Running the UI

Bowtie has a frontend interface which can be used to view or query results of Bowtie test runs.

A hosted version of this UI will live at https://bowtie.report/. If you are making changes to the UI, you can run it locally by:

- ensuring you have `node` and `npm` installed
- running `npm start` in the `frontend/` directory within your repository checkout of Bowtie

3.4 For Implementers

If you have a new (or not so new) implementation which Bowtie doesn't yet support, contributing support for your implementation is extremely useful.

See the *harness tutorial* for details on writing a harness for your implementation, and please do send a PR to add it!

3.5 Proposing Changes

If you suspect you may have found a bug, feel free to file an issue after checking whether it is a known issue. Of course a pull request to fix the bug is very welcome. Ideally, all pull requests (particularly ones that aren't frontend-focused) should come with tests to ensure the changes work and continue to work.

Continuous integration via GitHub actions should run to indicate whether your change passes both the test suite as well as linters. Please ensure it passes, or indicate in a comment if you believe it fails spuriously.

Please discuss any larger changes ahead of time for the sake of your own time! Improvements are very welcome, but large pull requests can be difficult to review or may be worth discussing design decisions before investing a lot of effort. You're welcome to suggest a change in an issue and thereby get some initial feedback before embarking on an effort that may not get merged.

3.6 Improving the Documentation?

Writing good documentation is challenging both to prioritize and to do well.

Any help you may have would be great, especially if you're a beginner who's struggled to understand Bowtie.

Documentation is written in Sphinx-flavored reStructuredText, so you'll want to familiarize yourself a bit with Sphinx.

Feel free to file issues or pull requests as well if you wish something was better documented, as this too can help prioritize.

CHAPTER

FOUR

USING BOWTIE IN GITHUB ACTIONS

Bowtie can be used from within GitHub Actions by using it in a GitHub workflow step. For example:

```yaml
name: Run Bowtie
on: [push]

jobs:
  bowtie:
    runs-on: ubuntu-latest

    steps:
      - name: Install Bowtie
        uses: bowtie-json-schema/bowtie@v2023.08.9
```

You will likely wish to use the latest version of Bowtie available.

See also:

Workflow Syntax for GitHub Actions

for full details on writing GitHub Actions workflows

Once you have installed it, the *Bowtie CLI* will be available in successive run steps. Most commonly, you can use it to validate an instance (some data) using a specific JSON Schema implementation by adding:

```yaml
- name: Validate Schema
  run: bowtie validate -i lua-jsonschema schema.json instance.json
```

replacing `lua-jsonschema` and the filenames with your implementation and schema of choice. For full details on the commands available, see the *CLI documentation*.

A fully working example of the above code can also be found here.

CHAPTER

FIVE

INSTALLATION

GitPod

You can use Bowtie immediately without installing it!

Click below to use it within GitPod, where you'll have immediate access to the Bowtie CLI and all of its supported implementations.

5.1 Via Homebrew (macOS or Linuxbrew)

Bowtie is available in a tap which is located here, and can be installed via:

```
brew install bowtie-json-schema/tap/bowtie
```

5.2 As a `shiv` / `pyapp`

There is an experimental `shiv` of Bowtie published to GitHub on each release.

You can find the latest one here.

Once downloading it, run `chmod +x` on it and you should be able to use it as-is if you have an existing Python installation.

If you use it (successfully or otherwise) please provide feedback.

5.3 Manual Installation via PyPI

Bowtie is written in Python, and uses a container runtime to execute JSON Schema implementations, so you'll need both Python and a suitable container runtime installed.

If you have no previous container runtime installed (e.g. Docker), follow the installation instructions for podman specific to your operating system. Ensure you've started a Podman VM if you are on macOS.

Then follow the pipx installation process to install `pipx`, and finally use it to install Bowtie via:

```
pipx install bowtie-json-schema
```

which should give you a working Bowtie installation, which you can check via:

```
bowtie --help
```

Further usage details of the command-line interface can be found *here*.

CHAPTER

SIX

EXECUTION

In general, executing Bowtie consists of providing 2 pieces of input:

- The names of one or more supported implementations to execute
- One or more test cases to run against these implementations (schemas, instances and optionally, expected validation results)

Given these, Bowtie will report on the result of executing each implementation against the input schema/instance pairs. If expected results are provided, it will compare the results produced against the expected ones, reporting on any implementations which differ from the expected output.

CHAPTER

SEVEN

USES

A key use of Bowtie is in executing as input the official test suite and comparing the results produced by implementations to the expected ones from the suite.

Bowtie however isn't limited to just the test cases in the test suite. It can be used to compare the validation results of any JSON Schema input across its supported implementations.

INDEX

Symbols

B

H

I

O

S

T

V