fq

jq for binary formats

Mattias Wadman

What is fq?

- Tool, language and decoders for working with data
- Inspried and based on jq
- Query and display data in a useful ways
- Interactive REPL with auto-completion
- Available for Linux, macOS, Windows, BSD
- Debugger for files

Why jq?

- CLI friendly syntax
  - Terse and composable `a | b | …`
  - Generators to iterate and recurse `.[] | …`, `.. | …`
- DSL to select and transform data
  - Superset of JSON `{a: [1,2+3,empty]}` ➡ `{"a": [1,5]}`
  - `{"a": 1, "b": 2}` ➡ `{sum: (.a+.b)}` ➡ `{"sum:" 3}`
- Purely functional langauge based on generators and backtracking
  - Conditionals, functions, bindings, special forms to collect and reduce etc
- Single filter run on each input
  - Can behave different using `inputs`, slurp, etc

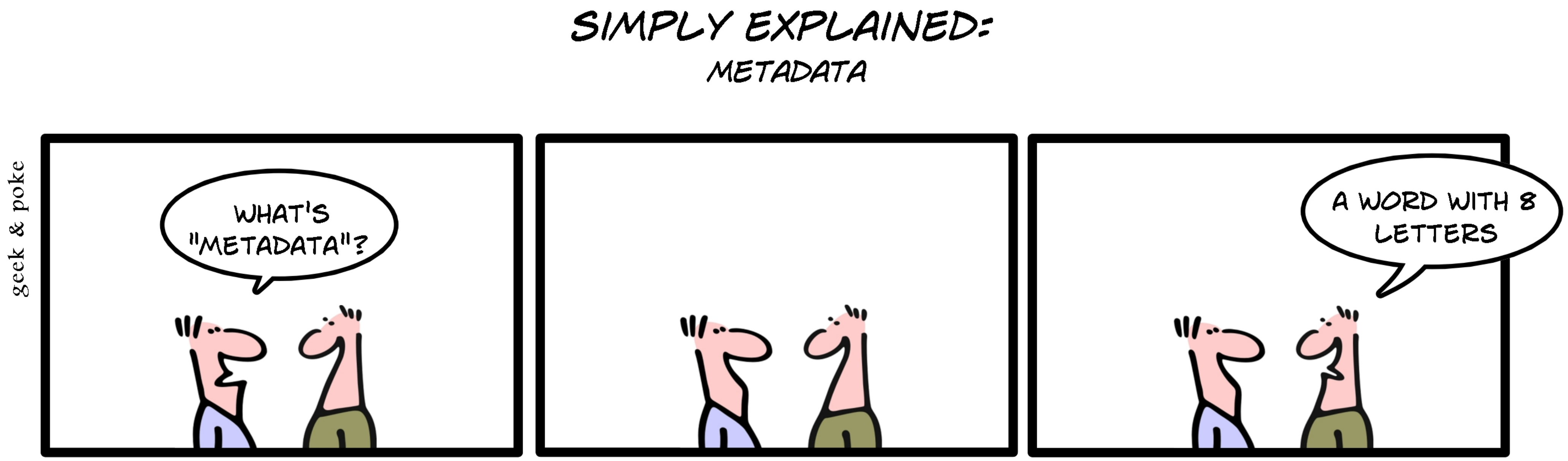
Supported formats

- Currently 113 formats
- Media containers (MP3, MP4, Ogg, FLAC, Matroska, PNG, JPEG, Exif, ...)
    - Some demux and decode samples (AAC, NALU, h264, MP3 frames, ...)
- Executables (ELF, Macho, Wasm)
- Archives and compression (ZIP, Tar, gz, bz2, ...)
- Network protocols (PCAP, Ethernet, UDP, TCP, IPV4, ...)
    - TCP reassembly and IPv4 defragmentation
- Serialization formats (MsgPack, ASN1 BER, CBOR, ProtoBuf, bplist, ...)
- Can also decode/encode text formats (YAML, XML, HTML, Toml, CSV, URLs, ...)

Decode in what way?

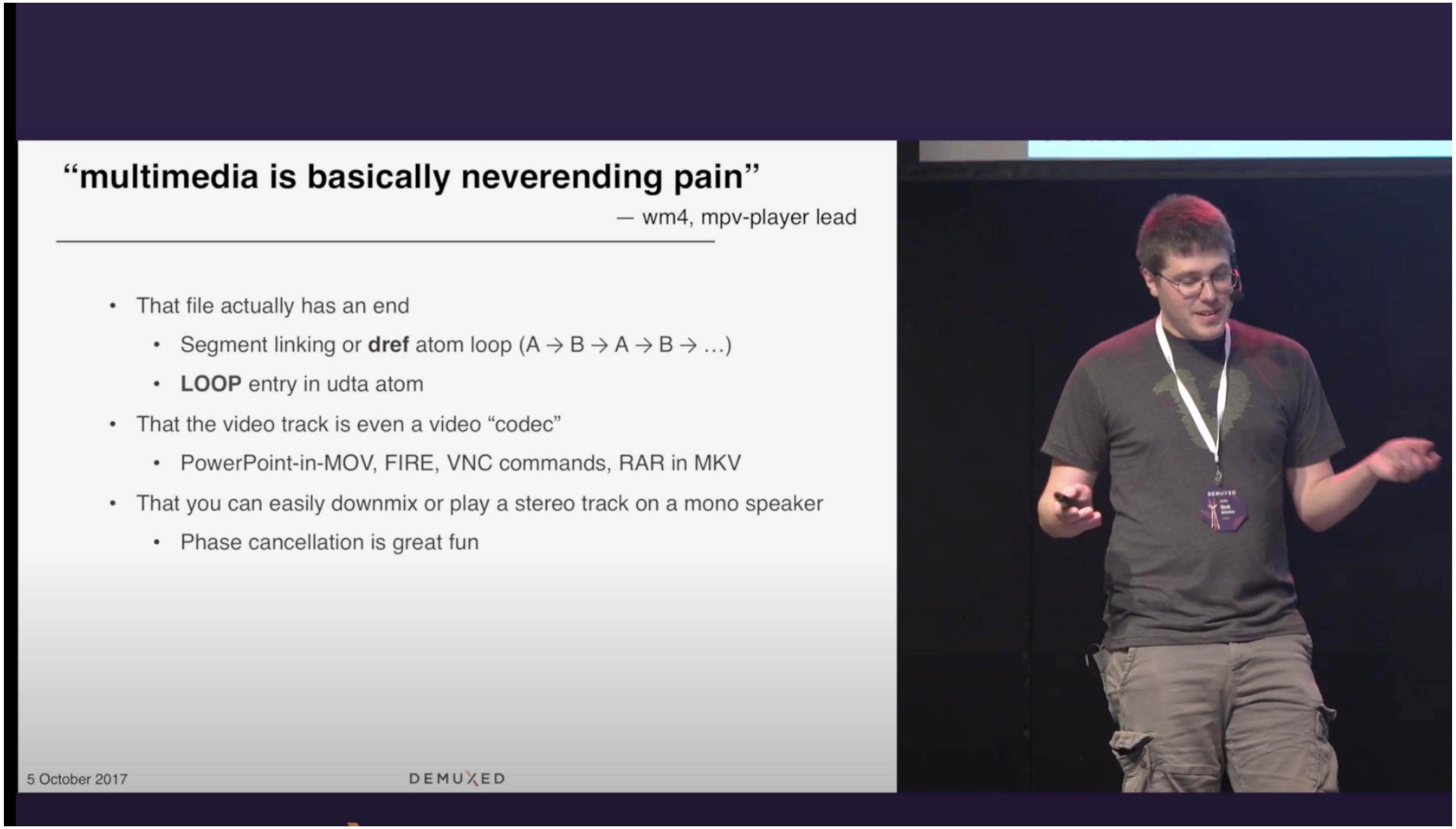
- A format decoder produces a JSON compatible structure
  - Each value has a bit range and optional description, mapping etc
- For media usually decode most things except the actual media
- For other formats more or less everything
- Some formats broken down into independent sub formats
  - `flac` into `flac_frame`, `flac_metadatablock`, `flac_streaminfo`, …
- Format can use other formats
  - `pcap` uses `ether8023_frame` that uses `ipv4_packet` that uses `tcp_segment`
  - `mp3` ➡ `id3v2` ➡ `jpeg` ➡ `exif` ➡ `icc_profile`
  - Can pass data between formats

My own use cases for fq



CC-BY-3.0 - https://geek-and-poke.com/geekandpoke/2010/4/17/meta.html

My own use cases for fq



“multimedia is basically neverending pain”

— wm4, mpv-player lead

- That file actually has an end
  - Segment linking or **dref** atom loop (A → B → A → B → …)
  - **LOOP** entry in udta atom
- That the video track is even a video “codec”
  - PowerPoint-in-MOV, FIRE, VNC commands, RAR in MKV
- That you can easily downmix or play a stereo track on a mono speaker
  - Phase cancellation is great fun

5 October 2017 DEMUXED

Derek Buitenhuis - Things Developers Believe About Video Files (Proven Wrong by User Uploads)(https://www.youtube.com/watch?v=cRSO3RtUOOk)

My own use cases for fq

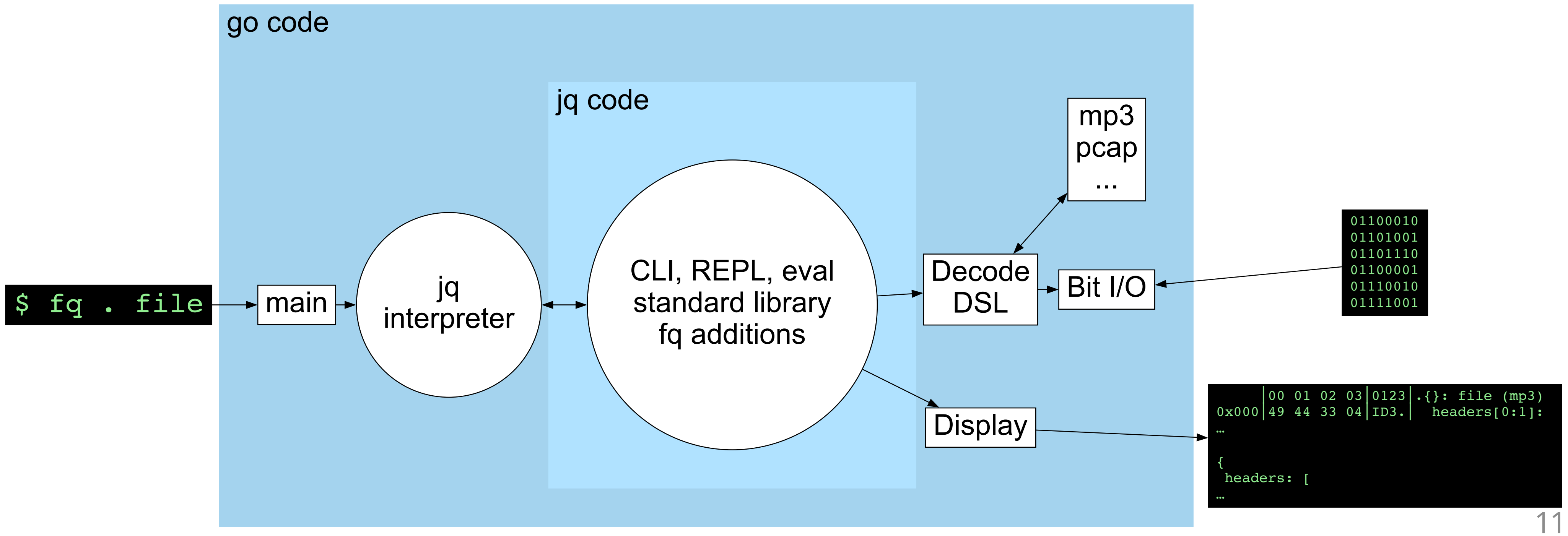
- Debug, query and assist when working with media files
- Show overview to look for unusual value or structures
- Sample tables, timestamps, decoder configuration, ...
- Find broken media sample
- Automate and aggregate over multiple files

What it can't do

- Very little encoding support, focus on decoding
  - Not clear how it would work (mov.c + movenc.c in ffmpeg ~17000 lines of dense C)
  - Tip! See next talks by Petr and Jose, both will touch on this I think.
- Can decode broken things but can't magically repair
  - Makes it possible but you probably need deep knowledge
- Decoders in runtime
  - Want to look into decoders in jq
  - Something declarative like Kaitai struct would be nice
  - Currently can slice and construct new binaries

Why go?

- Previous experience with go
- Two mature jq implementation
  - Original jq in C
  - gojq in go
- Robust, complex and safe decoders in C? 🤔
  - Probably need a VM etc
- Great tooling, fast builds, static binaries, cross platform



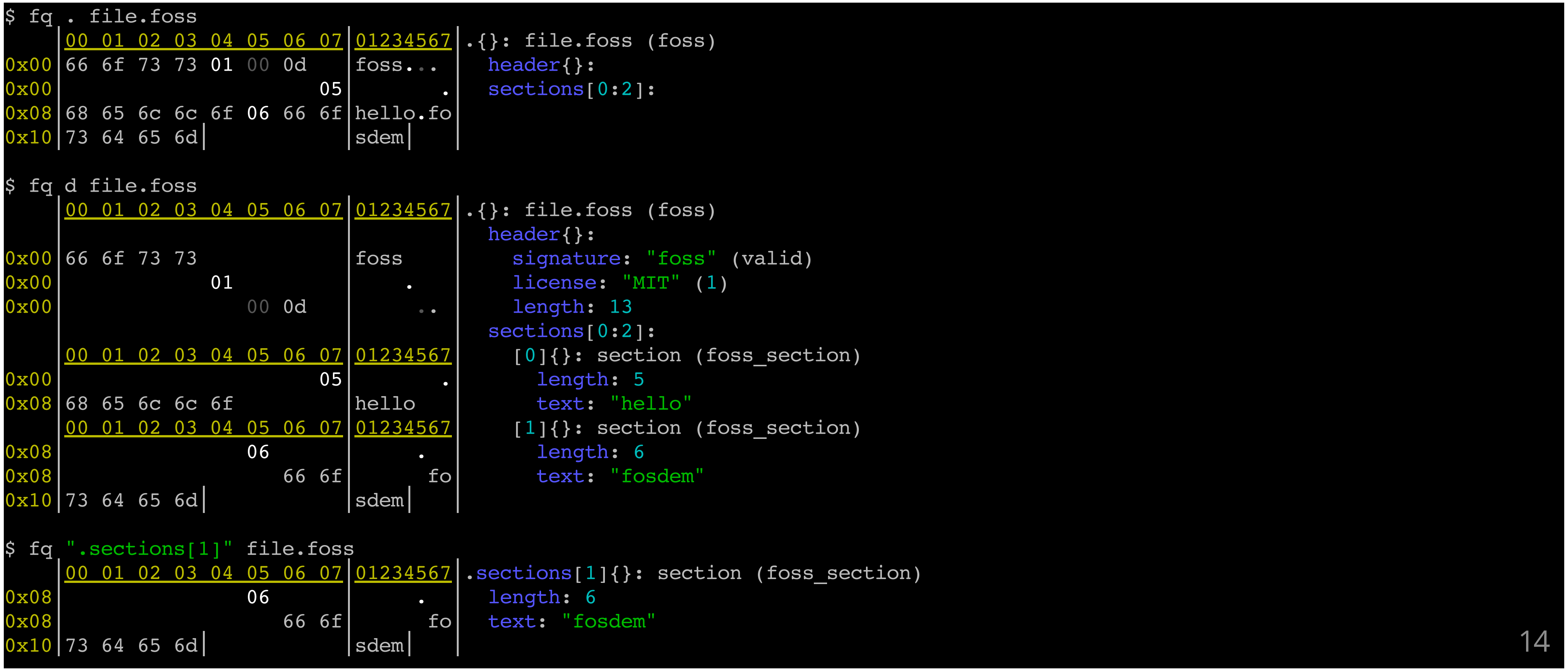
```
$ fq . file
```

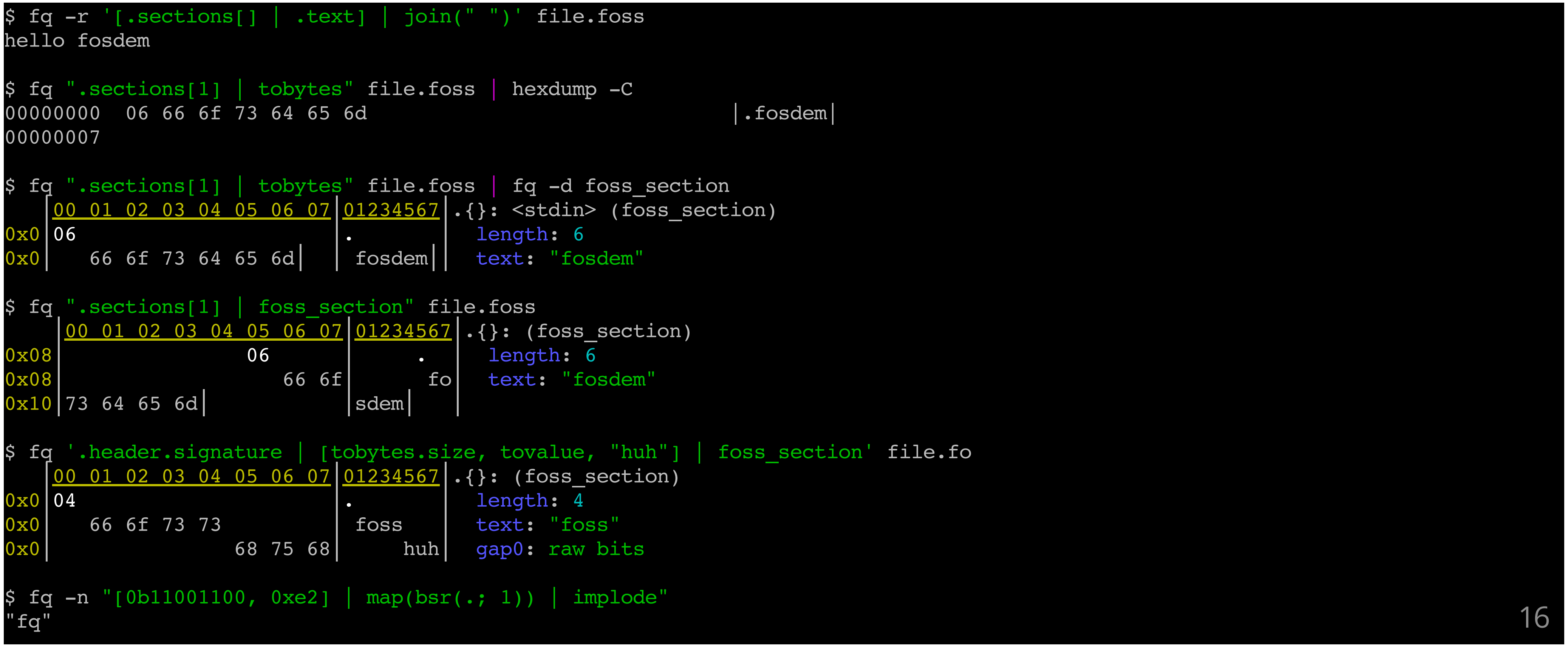
```
01100010
01101001
01101110
01100001
01110010
01111001
```

```
     |00 01 02 03|0123|.{}: file (mp3)
0x000|49 44 33 04|ID3.|  headers[0:1]:
…

{
 headers: [
…
```

```
def input:
  ( next_filename                  # next filename from global state
  | open                           # open it for reading
  | decode                         # decode using decode DSL
  );

def inputs: repeat(input);         # keep calling input

def main:
  ( parse_arguments as $opts
  | setup_some_global_state
  | if $opts.repl then repl
    else
      ( inputs                     # outputs each decoded value
      | eval($opts.filter)         # eval filter given as argument
      | display                    # show tree, JSON, binary etc
      )
    end
  );
```

```
func decodeFoss(d *decode.D, in any) any {
	var length uint64
	d.FieldStruct("header", func(d *decode.D) {
		d.FieldUTF8("signature", 4, d.StrAssert("foss"))
		d.FieldU8("license", scalar.UintMapSymStr{
			0: "GPL",
			1: "MIT",
		})
		length = d.FieldU16("length")
	})
	d.FramedFn(int64(length)*8, func(d *decode.D) {
		d.FieldArray("sections", func(d *decode.D) {
			for !d.End() {
				d.FieldFormat("section", sectionFormat, nil)
			}
		})
	})
	return nil
}
```


```
$ fq . file.foss
    |00 01 02 03 04 05 06 07|01234567|.{}: file.foss (foss)
0x00|66 6f 73 73 01 00 0d   |foss... |  header{}:
0x00|                     05|       .|  sections[0:2]:
0x08|68 65 6c 6c 6f 06 66 6f|hello.fo|
0x10|73 64 65 6d|           |sdem|   |

$ fq d file.foss
    |00 01 02 03 04 05 06 07|01234567|.{}: file.foss (foss)
                                     |  header{}:
0x00|66 6f 73 73            |foss    |    signature: "foss" (valid)
0x00|            01         |    .   |    license: "MIT" (1)
0x00|               00 0d   |     .. |    length: 13
                                     |  sections[0:2]:
    |00 01 02 03 04 05 06 07|01234567|    [0]{}: section (foss_section)
0x00|                     05|       .|      length: 5
0x08|68 65 6c 6c 6f         |hello   |      text: "hello"
    |00 01 02 03 04 05 06 07|01234567|    [1]{}: section (foss_section)
0x08|               06      |     .  |      length: 6
0x08|                  66 6f|      fo|      text: "fosdem"
0x10|73 64 65 6d|           |sdem|   |

$ fq ".sections[1]" file.foss
    |00 01 02 03 04 05 06 07|01234567|.sections[1]{}: section (foss_section)
0x08|               06      |     .  |  length: 6
0x08|                  66 6f|      fo|  text: "fosdem"
0x10|73 64 65 6d|           |sdem|   |
$ fq tovalue file
{
  "header": {
    "length": 13,
    "license": "MIT",
    "signature": "foss"
  },
  "sections": [
    {
      "length": 5,
      "text": "hello"
    },
    {
      "length": 6,
      "text": "fosdem"
    }
  ]
}

$ fq torepr file
{
  "sections": [
    "hello",
    "fosdem"
  ],
  "type": null
}
```



```
$ fq -r '[.sections[] | .text] | join(" ")' file.foss
hello fosdem

$ fq ".sections[1] | tobytes" file.foss | hexdump -C
00000000  06 66 6f 73 64 65 6d                              |.fosdem|
00000007

$ fq ".sections[1] | tobytes" file.foss | fq -d foss_section
   |00 01 02 03 04 05 06 07|01234567|.{}: <stdin> (foss_section)
0x0|06                     |.       |  length: 6
0x0|   66 6f 73 64 65 6d|  | fosdem||  text: "fosdem"

$ fq ".sections[1] | foss_section" file.foss
    |00 01 02 03 04 05 06 07|01234567|.{}: (foss_section)
0x08|               06      |     .  |  length: 6
0x08|                  66 6f|      fo|  text: "fosdem"
0x10|73 64 65 6d|           |sdem|   |

$ fq '.header.signature | [tobytes.size, tovalue, "huh"] | foss_section' file.fo
   |00 01 02 03 04 05 06 07|01234567|.{}: (foss_section)
0x0|04                     |.       |  length: 4
0x0|   66 6f 73 73         | foss   |  text: "foss"
0x0|               68 75 68|     huh|  gap0: raw bits

$ fq -n "[0b11001100, 0xe2] | map(bsr(.; 1)) | implode"
"fq"
```

Thanks and useful tools

- itchyny for gojq
- Stephen Dolan and others for jq
- HexFiend
- GNU poke
- Kaitai struct
- Wireshark
- vscode-jq (https://github.com/wader/vscode-jq)
- jq-lsp (https://github.com/wader/jq-lsp)

Thank you

jq for binary formats

Mattias Wadman
mattias.wadman@gmail.com (mailto:mattias.wadman@gmail.com)
https://github.com/wader/fq (https://github.com/wader/fq)
https://fosstodon.org/@wader (https://fosstodon.org/@wader)