Towards a more expressive and introspectable QEMU command line

Markus Armbruster <armbru@redhat.com>

KVM Forum 2017

Part I
Things we want from the command line

A simple example

You could run QEMU like this:

```
$ qemu -s -machine usb=on,accel=kvm disk.qcow2
```

Observe:

- Options, with and without arguments, as usual
- Complex argument of the form *key*=*value*,...

A real-world example

```
$ /usr/bin/qemu-system-x86_64 -machine accel=kvm -name boxes-unknown -S -machine pc-i440fx-1.6,accel=kvm,usb=off -cpu \
Penryn -m 3115 -realtime mlock=off -smp 4,sockets=1,cores=4,threads=1 -uuid 8bd53789-adab-484f-8c53-a6df9d5f1dbf -no-u\
ser-config -nodefaults -chardev socket,id=charmonitor,path=/home/guillaume/.config/libvirt/qemu/lib/boxes-unknown.moni\
tor,server,nowait -mon chardev=charmonitor,id=monitor,mode=control -rtc base=utc,driftfix=slew -global kvm-pit.lost_ti\
ck_policy=discard -no-shutdown -global PIIX4_PM.disable_s3=1 -global PIIX4_PM.disable_s4=1 -boot strict=on -device ich\
9-usb-ehci1,id=usb,bus=pci.0,addr=0x5.0x7 -device ich9-usb-uhci1,masterbus=usb.0,firstport=0,bus=pci.0,multifunction=o\
n,addr=0x5 -device ich9-usb-uhci2,masterbus=usb.0,firstport=2,bus=pci.0,addr=0x5.0x1-device ich9-usb-uhci3,masterbus=u\
sb.0,firstport=4,bus=pci.0,addr=0x5.0x2 -device virtio-serial-pci,id=virtio-serial0,bus=pci.0,addr=0x6 -device usb-cci\
d,id=ccid0 -drive file=/home/guillaume/.local/share/gnome-boxes/images/boxes-unknown,if=none,id=drive-ide0-0-0,format=\
qcow2,cache=none -device ide-hd,bus=ide.0,unit=0,drive=drive-ide0-0-0,id=ide0-0-0,bootindex=1 -drive if=none,id=drive-\
ide0-1-0,readonly=on,format=raw -device ide-cd,bus=ide.1,unit=0,drive=drive-ide0-1-0,id=ide0-1-0 -netdev tap,fd=23,id=\
hostnet0 -device rtl8139,netdev=hostnet0,id=net0,mac=52:54:00:db:56:54,bus=pci.0,addr=0x3 -chardev spicevmc,id=charsma\
rtcard0,name=smartcard -device ccid-card-passthru,chardev=charsmartcard0,id=smartcard0,bus=ccid0.0 -chardev pty,id=cha\
rserial0 -device isa-serial,chardev=charserial0,id=serial0 -chardev spicevmc,id=charchannel0,name=vdagent -device virt\
serialport,bus=virtio-serial0.0,nr=1,chardev=charchannel0,id=channel0,name=com.redhat.spice.0 -device usb-tablet,id=in\
put0 -spice port=5901,addr=127.0.0.1,disable-ticketing,image-compression=off,seamless-migration=on -device qxl-vga,id=\
video0,ram_size=67108864,vram_size=67108864,vgamem_mb=16,bus=pci.0,addr=0x2 -device AC97,id=sound0,bus=pci.0,addr=0x4 \
-chardev spicevmc,id=charredir0,name=usbredir -device usb-redir,chardev=charredir0,id=redir0 -chardev spicevmc,id=char\
redir1,name=usbredir -device usb-redir,chardev=charredir1,id=redir1 -chardev spicevmc,id=charredir2,name=usbredir -dev\
ice usb-redir,chardev=charredir2,id=redir2 -chardev spicevmc,id=charredir3,name=usbredir -device usb-redir,chardev=cha\
rredir3,id=redir3 -incoming fd:20 -device virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x7 -msg timestamp=on
```

We clearly push CLI beyond its intended use...

Wanted: config files

Some use cases are better served by config files:

```
$ qemu-system-x86_64 -readconfig vm1.cfg
```

Everything CLI should also work in config files

Another config interface: QMP

QEMU Monitor Protocol (QMP):

```
QMP> {"execute": "blockdev-add",
      "arguments": {"node-name": "node1",
                    "driver": "file",
                    "filename": "tmp.img"}}
{"return": {}}
```

Observe:

- Commands and responses are JSON objects

Why *two* config interfaces?

Run-time reconfiguration must use QMP

Much initial configuration uses CLI, because...

Why *two* config interfaces?

Run-time reconfiguration must use QMP

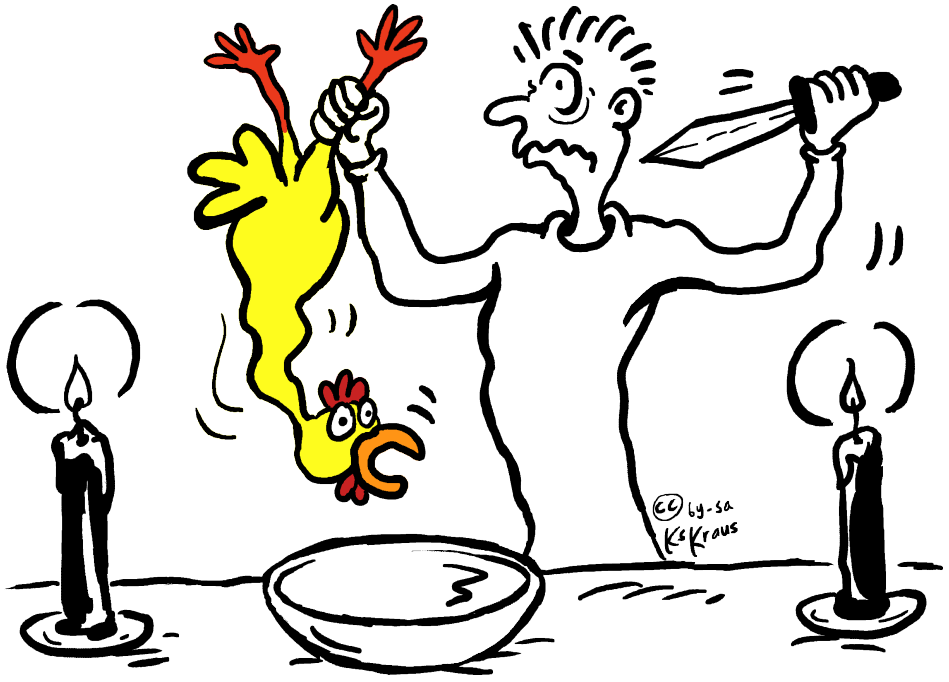
Much initial configuration uses CLI, because

- we've always done it this way (and we turn like a tanker)

Why *two* config interfaces?

Run-time reconfiguration must use QMP

Much initial configuration uses CLI, because



- we've always done it this way (and we turn like a tanker)
- we're *devoted* to backward compatibility

Wanted: equality

Some configuration is needed both in CLI and QMP
(e.g. `-chardev` & `chardev-add`, `-object` & `object-add`)

Our infrastucture should support this:

- CLI and QMP need to be equally expressive
  - QMP needs to express CLI's *key*=*value*,...
  - CLI needs to express QMP's JSON objects

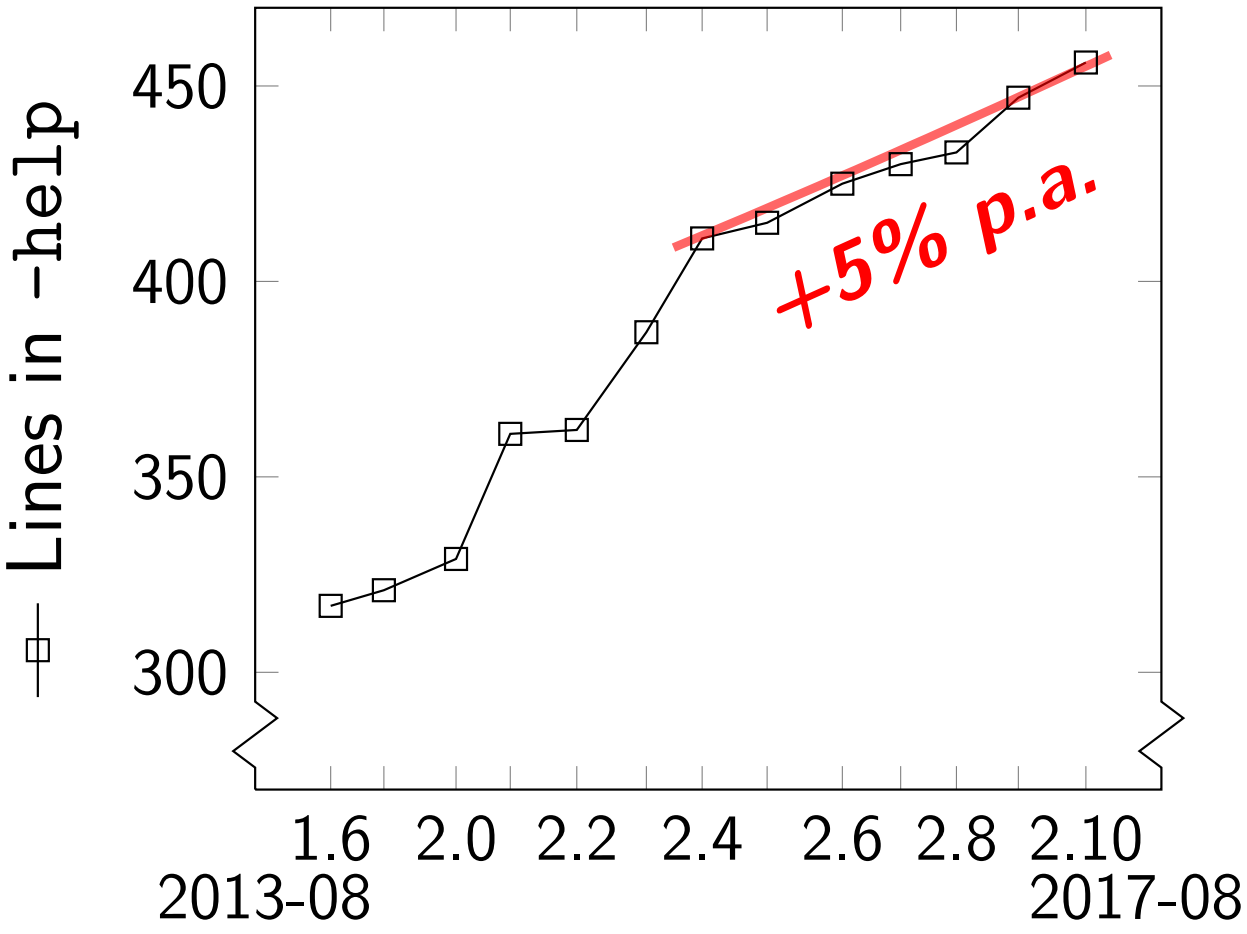
Wanted: equality

Some configuration is needed both in CLI and QMP
(e.g. `-chardev` & `chardev-add`, `-object` & `object-add`)

Our infrastucture should support this:

- CLI and QMP need to be equally expressive
  - QMP needs to express CLI's *key*=*value*,...
  - CLI needs to express QMP's JSON objects
- Want to drive single C interface with equal ease

CLI evolves constantly



Wanted: interface introspection

Programs interfacing with QEMU need to know:

- What options are available?

  Example: does this QEMU support `-blockdev`?
- What keys does an option support?

  Example: does `-spice` support `unix`)?
- What values does a key support?

  Example: does `-blockdev` support `driver=gluster`?

⇝ CLI needs to support introspection

Part II
Are these needs met? (TLDR: nope)

CLI option definition

Options are defined like this:

```
DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    "-msg timestamp[=on|off]\n"
    "       change the format of messages\n"
    "       on|off controls leading timestamps\n",
    QEMU_ARCH_ALL)
STEXI
@item -msg timestamp[=on|off]
@findex -msg
prepend a timestamp to each log message.
ETEXI
```

CLI option definition

Options are defined like this:

```
DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    "-msg timestamp[=on|off]
          on|off controls leading timestamps\n",
    QEMU_ARCH_ALL)
STEXI
@item -msg timestamp[=on|off]
@findex -msg
prepend a timestamp to each log message.
ETEXI
```

Option -msg, has mandatory argument

CLI option definition

Options are defined like this:

```
DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    "-msg timestamp[=on|off]\n"
    "       change the format of messages\n"
    "       on|off controls leading timestamps\n",
    QEMU_ARCH_ALL)
STEXI
@item -msg timestamp[=on|off]
@findex -msg
prepend a timestamp to each log message.
ETEXI
```

Text for -help

CLI option definition

Options are defined like this:

```
DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    "-msg timestamp[=on|off]\n"
    "       change the format of messages\n"
    "       on|off controls leading timestamps\n",
    QEMU_ARCH_ALL)
STEXI
@item -msg timestamp[=on|off]
@findex -msg
prepend a timestamp to each log message.
ETEXI
```

Text for user manual

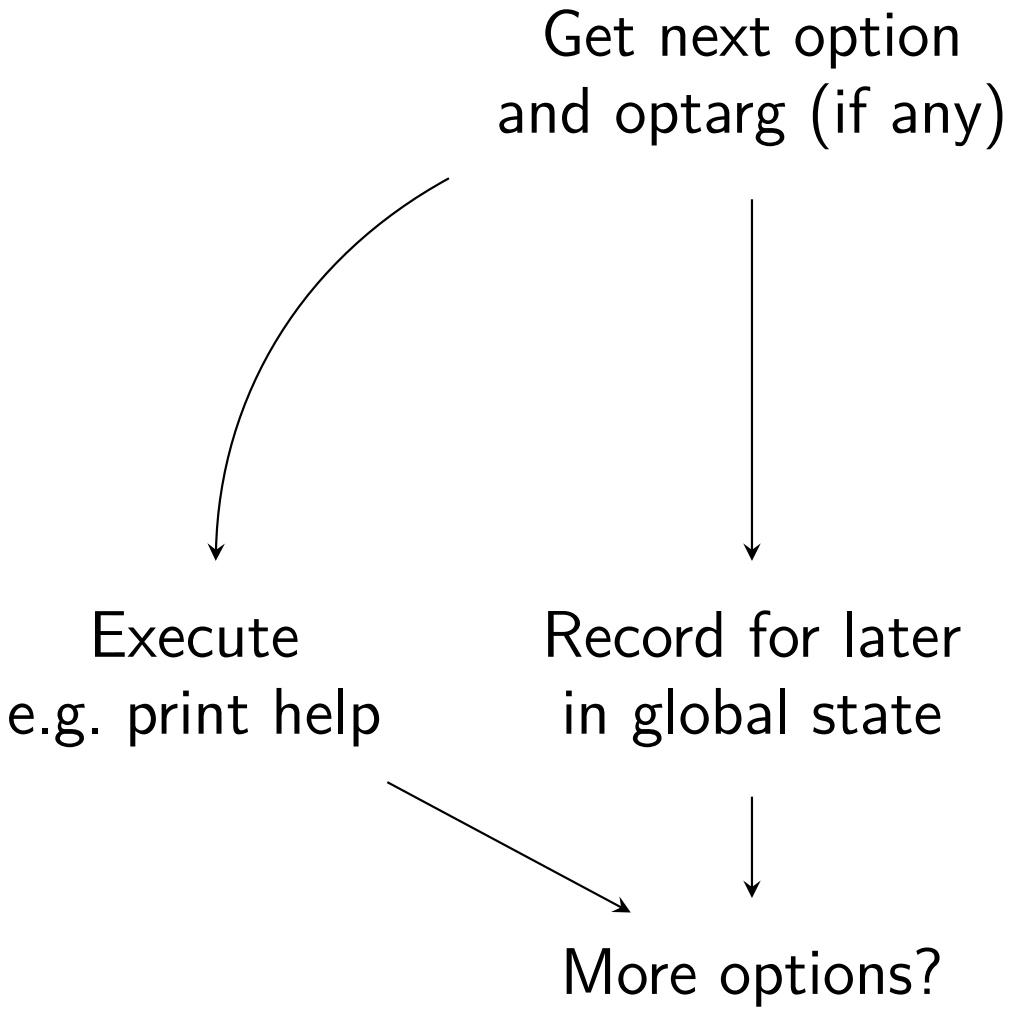
CLI option definition

Options are defined like this:

```
DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    "-msg timestamp[=on|off]\n"
    "       change the format of messages\n"
    "
    QEMU_A
STEXI
@item -msg timestamp[=on|off]
@findex -msg
prepend a timestamp to each log message.
ETEXI
```

Optarg format in help and manual text, but not in code

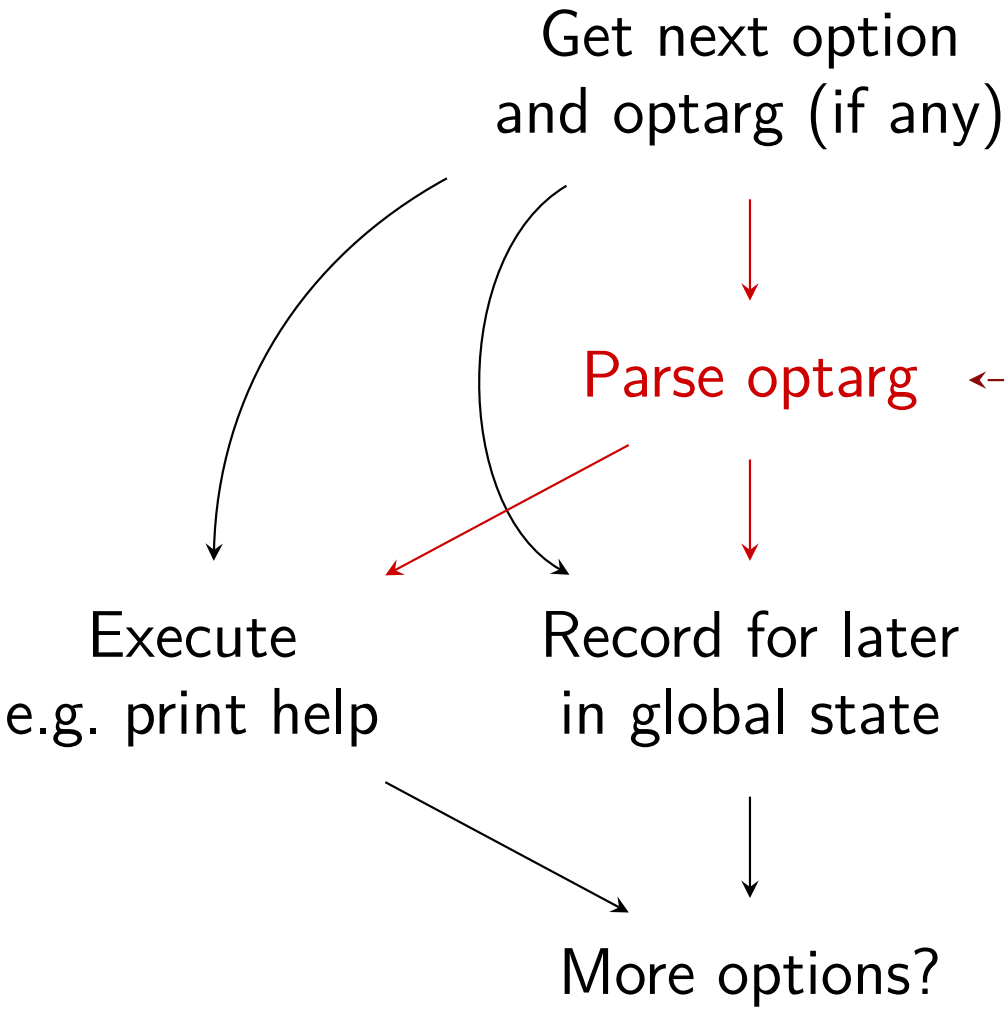
How we parse CLI



Simple optarg

`-help`, `-hda` *FILE*

How we parse CLI

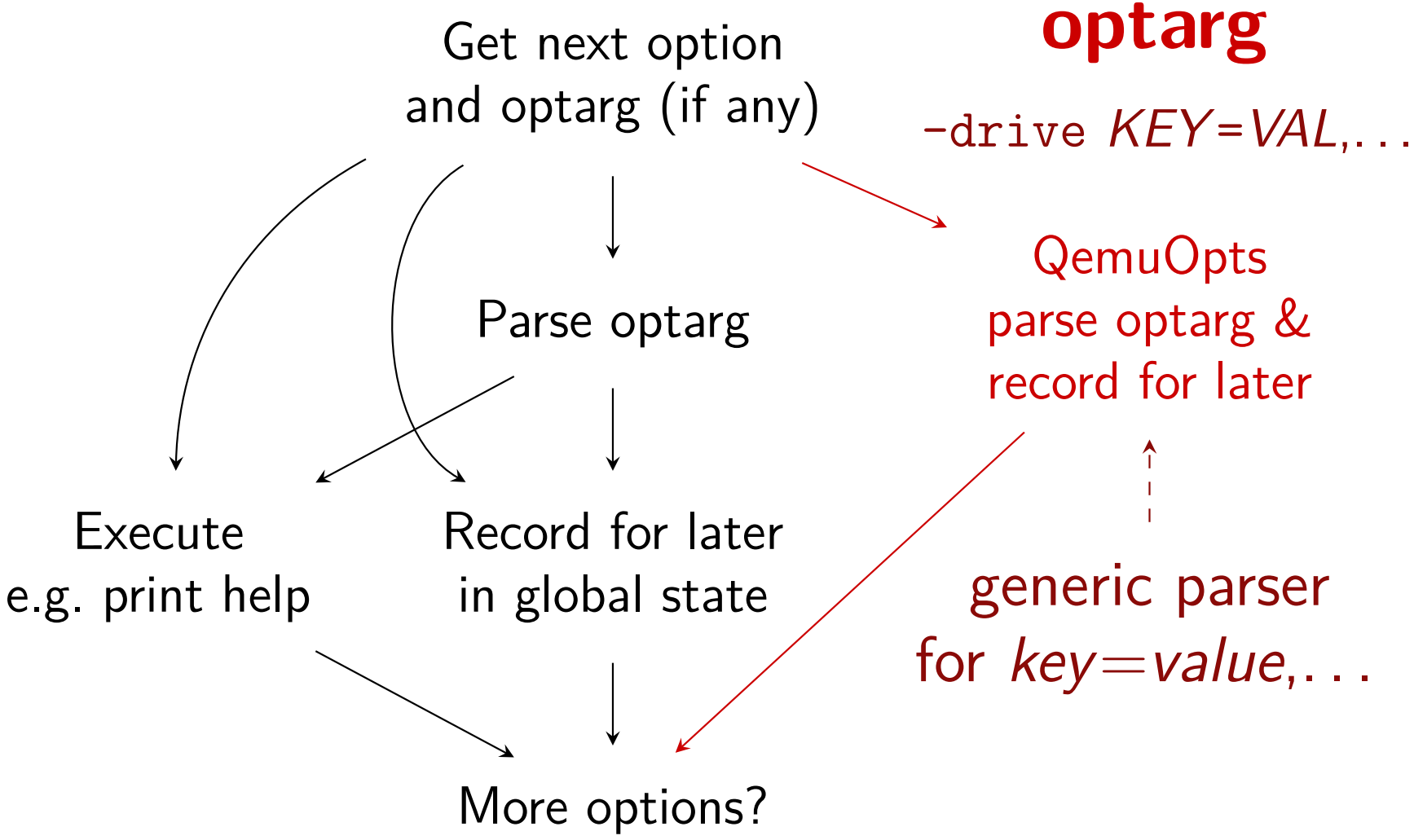


Complex optarg

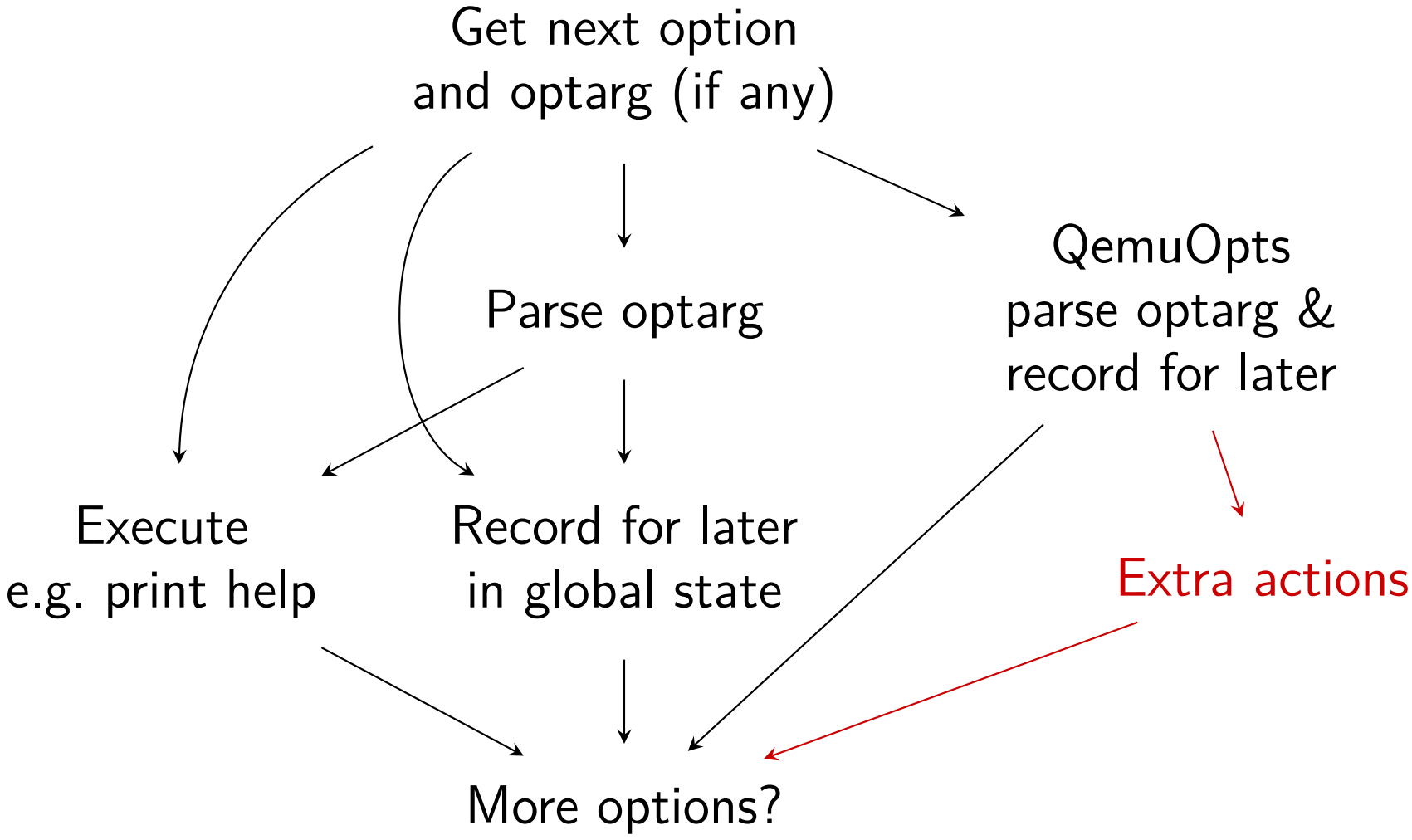
`-cpu` *CPU*,[-+]*FLAG*,...

20+ ad hoc parsers

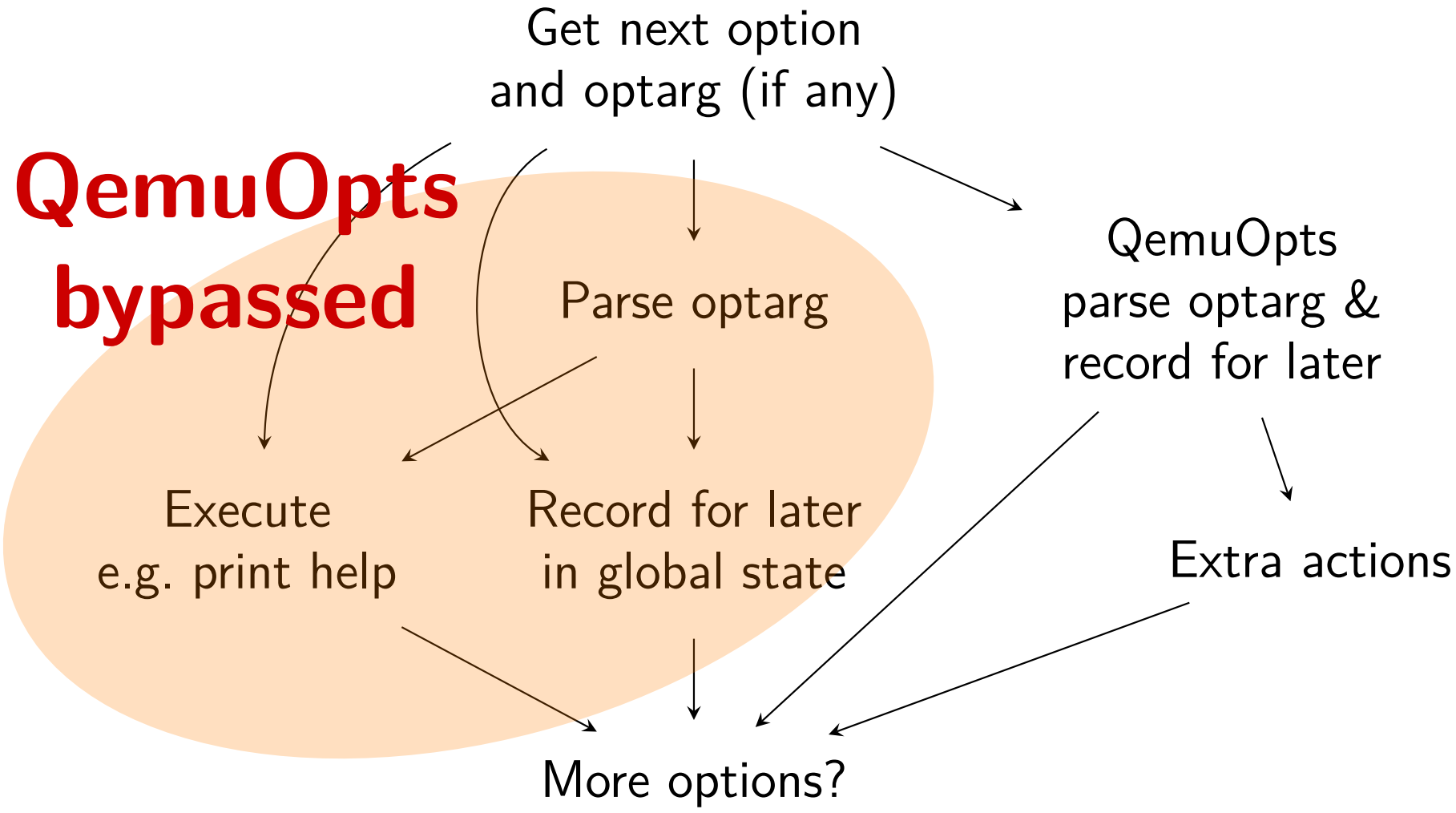
How we parse CLI



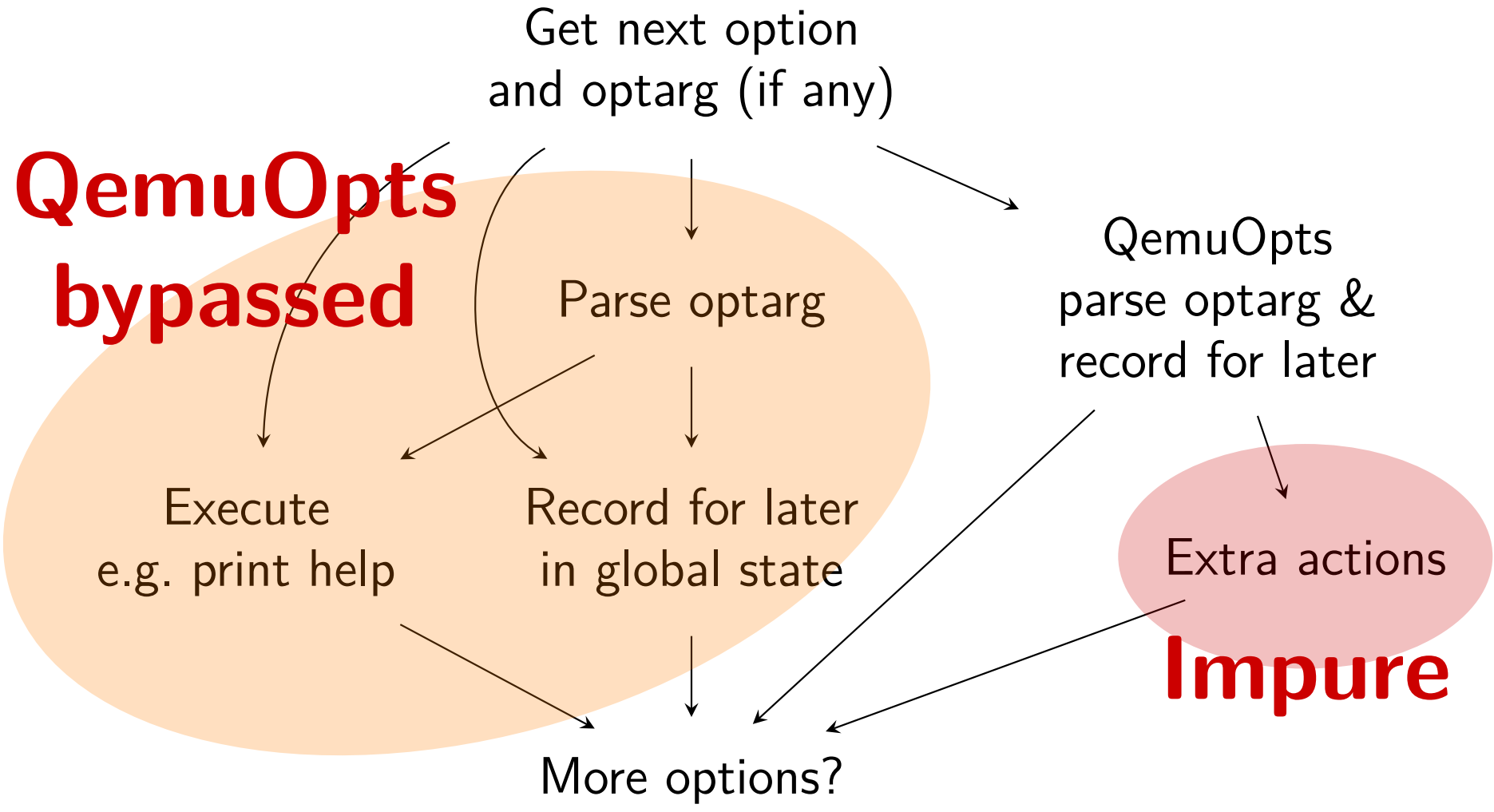
How we parse CLI



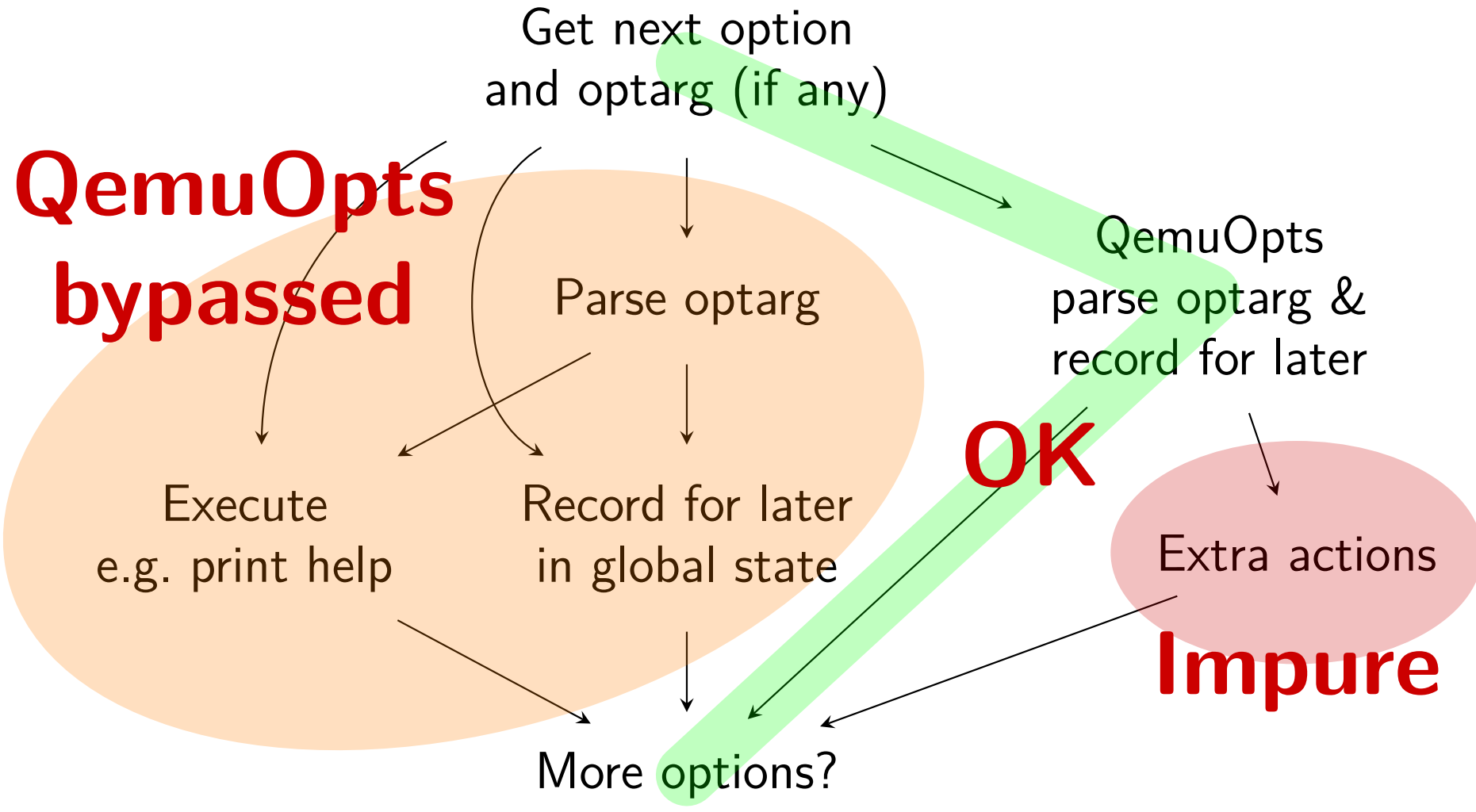
How we parse CLI



How we parse CLI



How we parse CLI



Impact on CLI config files

Config files apply to QemuOpts state:

- `-writeconfig` writes it out
- `-readconfig` reads it in

Impact of QemuOpts' sad state on config files:

- Okay: works
- Impure: broken (extra actions skipped)
- Bypassed: not supported

Config files work for one out of five options

Impact on introspection

Introspection is based on QemuOpts:

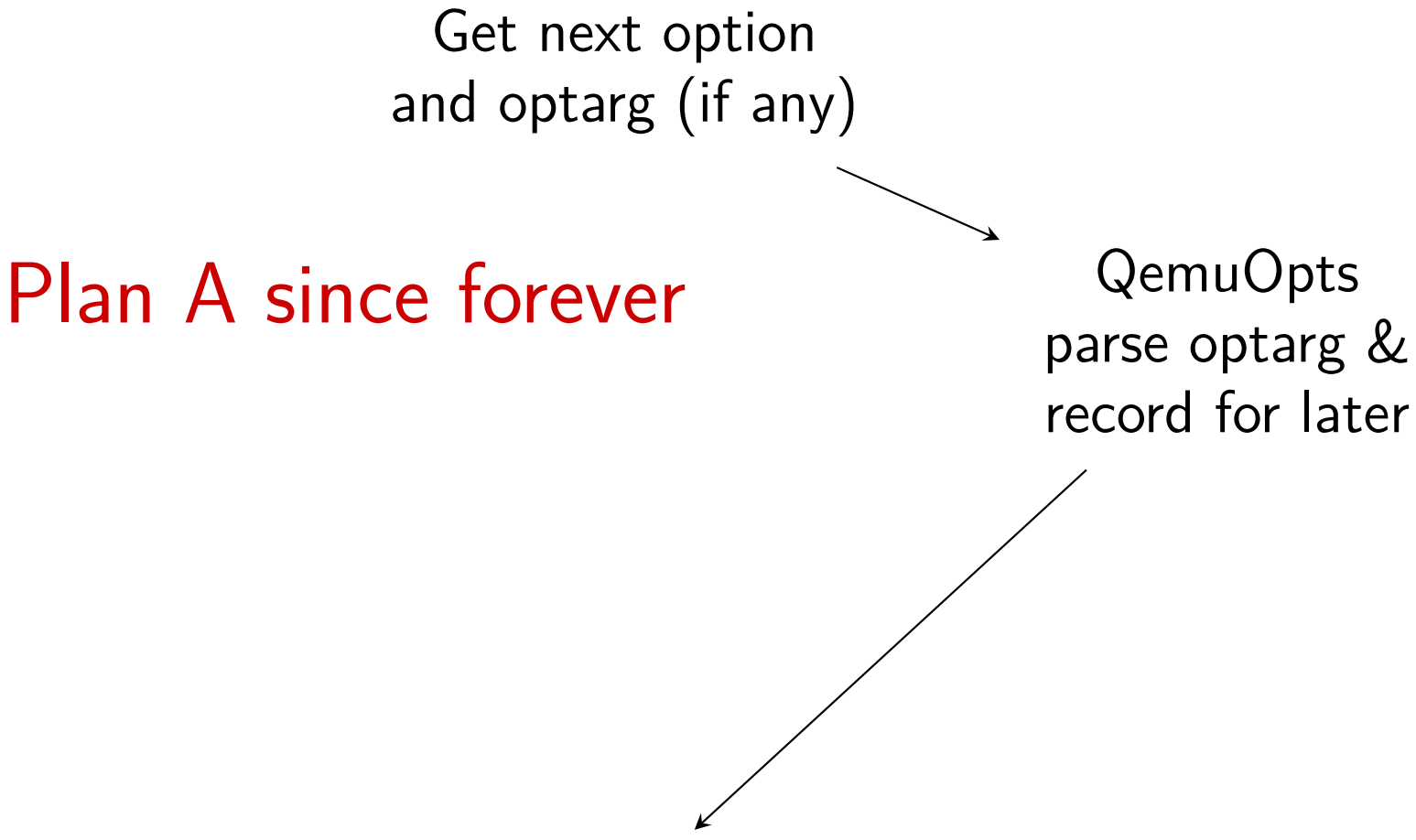
> QMP's `query-command-line-options` has no other source of truth

Impact on introspection:

- Okay: works
- Impure: works anyway
- Bypassed: not supported

Can introspect one out of five options

Fix by QemuOpts taking over?



Plan A since forever

Fix by QemuOpts taking over?

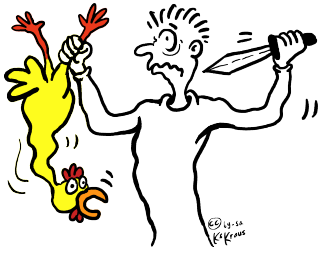
Get next option
and optarg (if any)

QemuOpts
parse optarg &
record for later

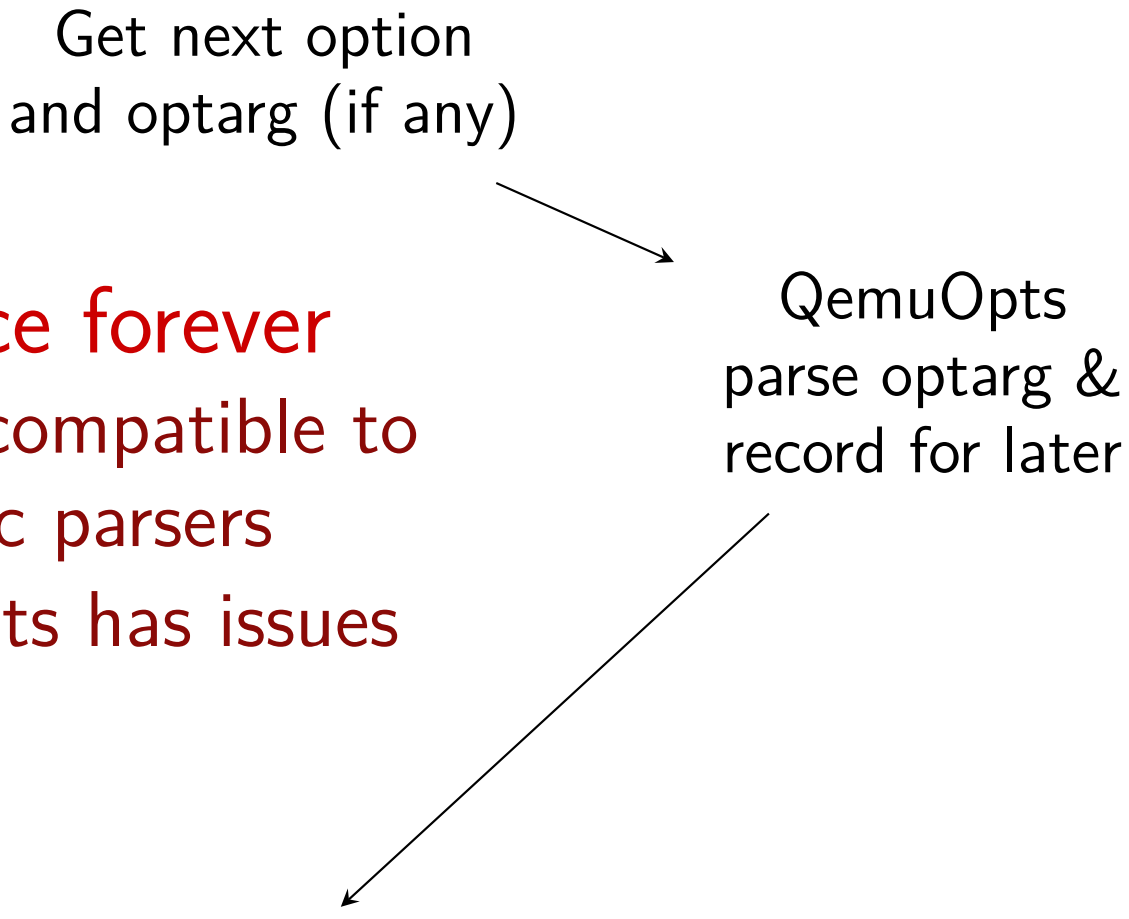
More options?

Plan A since forever

Problem: stay compatible to
20+ ad hoc parsers



Fix by QemuOpts taking over?



Plan A since forever

Problem: stay compatible to 20+ ad hoc parsers

Also: QemuOpts has issues

QemuOpts' data model

Derived from abstract *key*=*value*,... syntax:

- Keyword parameters, all optional
- Parameter values are strings

Encapsulated in type `QemuOpts`:

- Parameters are dynamically typed

Thrown in for convenience:

- Can restrict values to integer or bool

Stupidest model that could possibly work

Simple example: -msg

-msg has just one bool parameter timestamp

QemuOpts declaration:

```
static QemuOptsList qemu_msg_opts = {
    .name = "msg",
    [boilerplate omitted...]
    .desc = {
        {
            .name = "timestamp",
            .type = QEMU_OPT_BOOL,
        },
        { 0 }
    },
};
```

Simple example: -msg

-msg has just one bool parameter timestamp

QemuOpts declaration:

```
static QemuOptsList qemu_msg_opts = {
    .name = "msg",
    [boilerplate omitted...]
    .desc = {
        {
            .name = "timestamp",
            .type = QEMU_OPT_BOOL,
        },
        { 0 }
    },
};
```

Parameter name and type

Simple example: -msg

-msg has just one bool parameter timestamp

QemuOpts declaration:

```
static QemuOptsList qemu_msg_opts = {
    .name = "msg",
    [boilerplate omitted...]
    .desc = {
        {
            .name = "timestamp",
            .type = QEMU_OPT_BOOL,
        },
        { 0 }
    },
};
```

Note: option *argument* definition is separate from *option* definition

Next example: -numa

`-numa` has a mandatory parameter `type`
Additional parameters depend on value of `type`
(e.g. with `type=node`, we have parameter `nodeid`)

QemuOpts declaration:

```
static QemuOptsList qemu_numa_opts = {
    .name = "numa",
    [boilerplate omitted...]
    .desc = {
        { 0 }
    },
};
```

Next example: -numa

`-numa` has a mandatory parameter `type`
Additional parameters depend on value of `type`
(e.g. with `type=node`, we have parameter `nodeid`)

QemuOpts declaration:

```
static QemuOptsList qemu_numa_opts = {
    .name = "numa",
    [boilerplate omitted...]
    .desc = {
        { 0 }
    },
};
```

Best QemuOpts can do:
accept any key, with string value
(bye, bye introspection)

Code to parse -numa's optarg

```
QemuOptsList *list = qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg,
                                 [details...]);
[error out if !opts...]
const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
```

Code to parse -numa's optarg

Find the declaration

```
QemuOptsList *list = qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg,
                                 [details...]);
[error out if !opts...]
const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
```

Code to parse -numa's optarg

Parse optarg

```
= qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg,
                                 [details...]);
[error out if !opts...]
const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
```

Code to parse -numa's optarg

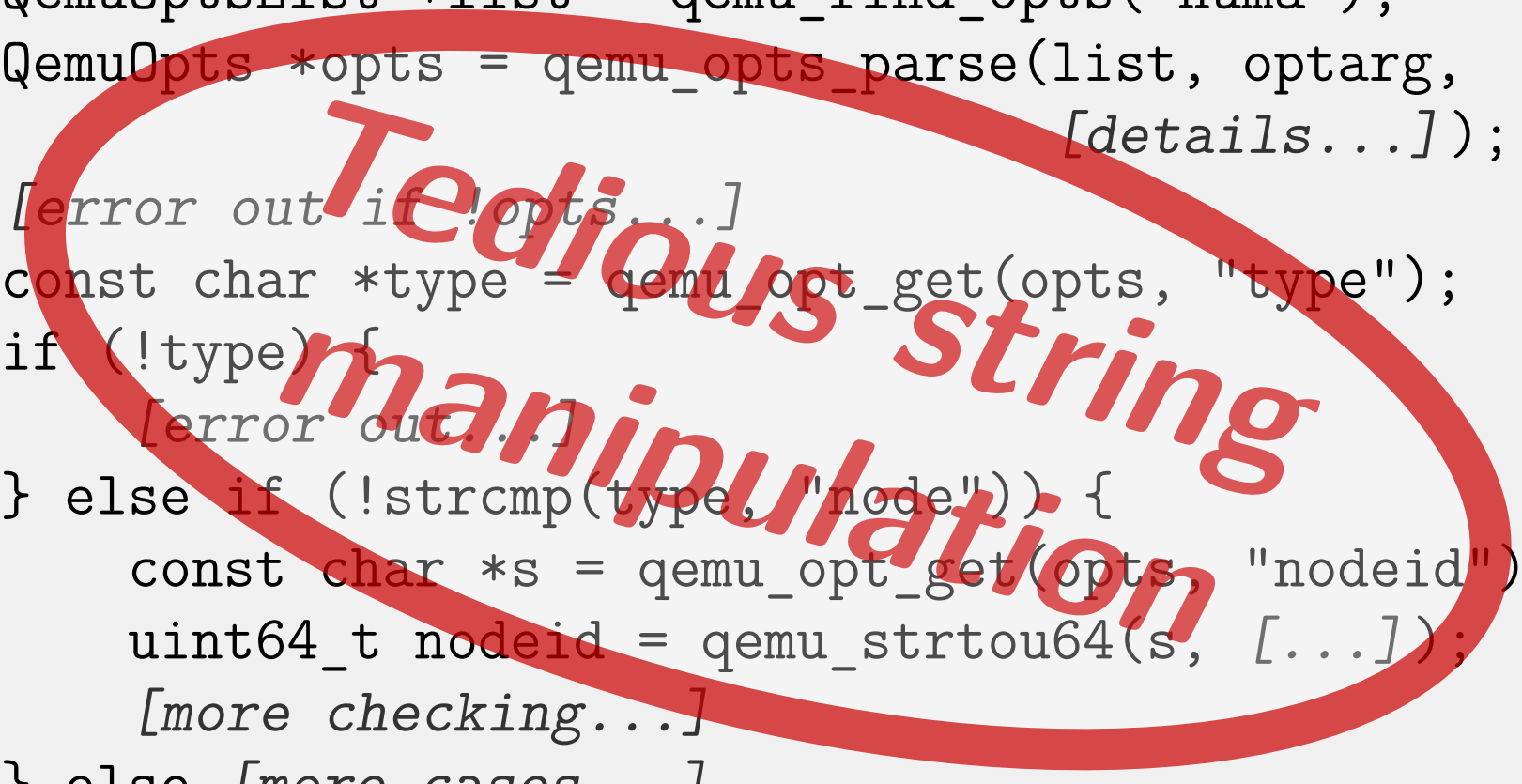
```
QemuOptsList *list = qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg,
                                 [details...]);
```

Get and check parameters

```
const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
```

Code to parse -numa's optarg

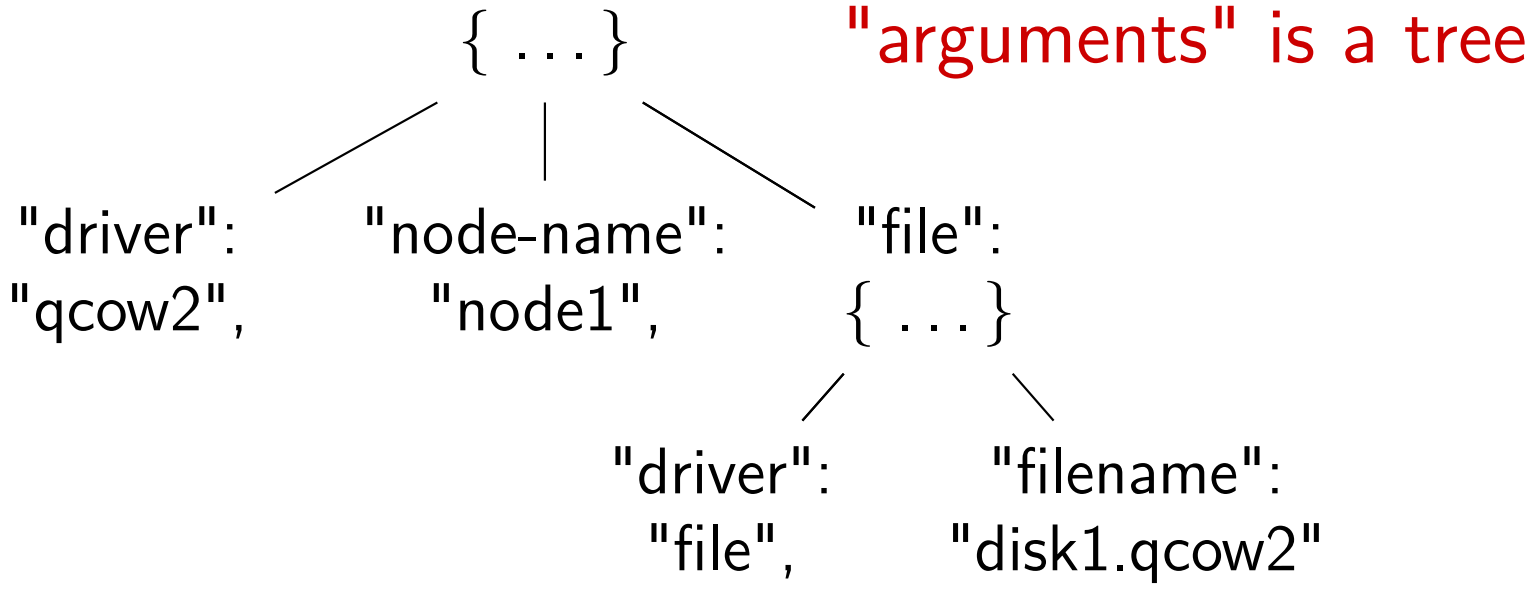
```
QemuOptsList *list = qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg,
                                 [details...]);
[error out if !opts...]
const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
```



Third example: -blockdev

`-blockdev` is like QMP `blockdev-add`

```
QMP> {"execute": "blockdev-add", "arguments": {
        "driver": "qcow2", "node-name": "node1",
        "file":{ "driver": "file",
                 "filename": "disk1.qcow2"}}}
```



Third example: -blockdev

`-blockdev` is like QMP `blockdev-add`

```
QMP> {"execute": "blockdev-add", "arguments": {
        "driver": "qcow2", "node-name": "node1",
        "file":{ "driver": "file",
                 "filename": "disk1.qcow2"}}}
```

But QemuOpts is by design flat...

Third example: -blockdev

`-blockdev` is like QMP `blockdev-add`

```
QMP> {"execute": "blockdev-add", "arguments": {
        "driver": "qcow2", "node-name": "node1",
        "file":{ "driver": "file",
                 "filename": "disk1.qcow2"}}}
```

But QemuOpts is by design flat...

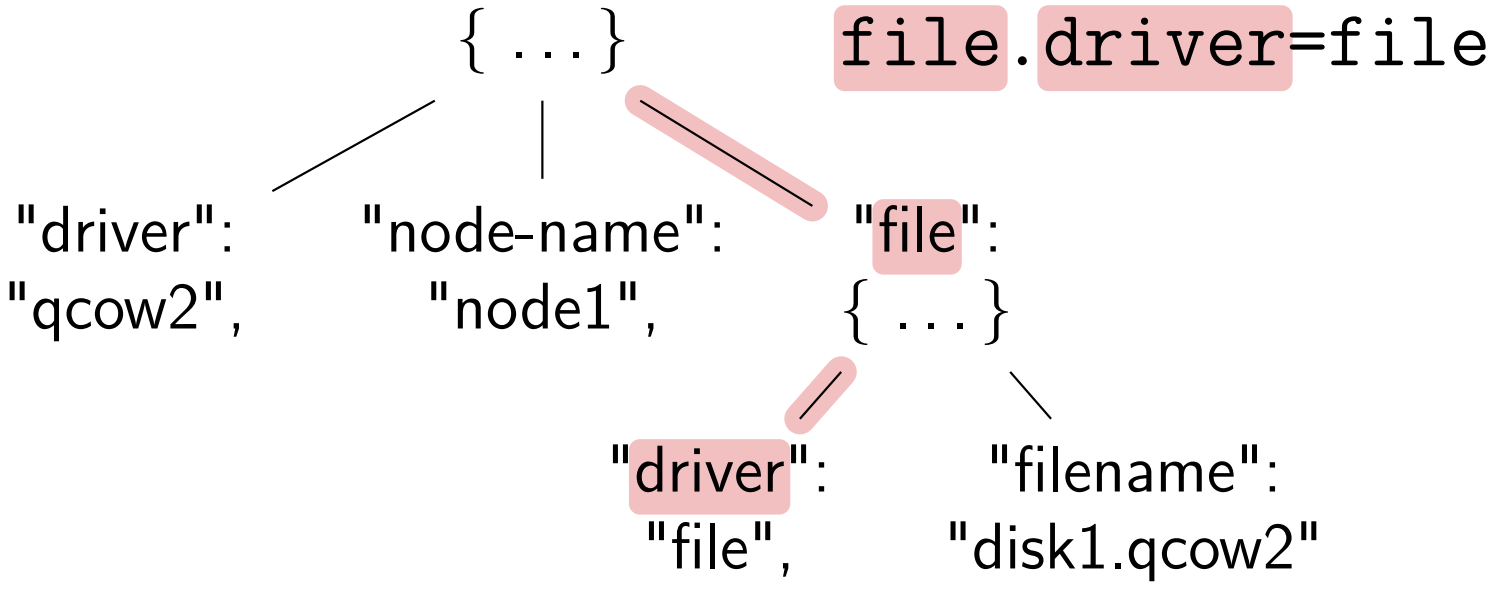
Flatten the arguments tree with dotted keys:

```
-blockdev driver=qcow2,node-name=node1,\
file.driver=file,file.filename=tmp.qcow2
```

Dotted keys in a nutshell

Basic idea:

- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf



Dotted keys in a nutshell

Basic idea:

- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:

- Bolted onto QemuOpts

  $\rightsquigarrow$ Bye, bye introspection, hello extra code

Dotted keys in a nutshell

Basic idea:

- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:

- Bolted onto QemuOpts
  $\rightsquigarrow$ Bye, bye introspection, hello extra code
- Inconsistent with *other* workarounds

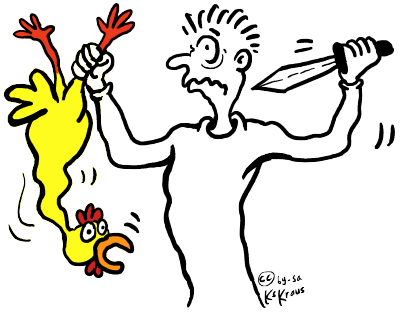
Dotted keys in a nutshell

Basic idea:

- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:

- Bolted onto QemuOpts
  $\rightsquigarrow$ Bye, bye introspection, hello extra code
- Inconsistent with *other* workarounds
  (to be kept for backward compatibility)



Dotted keys in a nutshell

Basic idea:

- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:

- Bolted onto QemuOpts
  $\rightsquigarrow$ Bye, bye introspection, hello extra code
- Inconsistent with *other* workarounds
  (to be kept for backward compatibility)
- Can express most, but not all trees

Dotted keys in a nutshell

Basic idea:

- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:

- Bolted onto QemuOpts
  $\rightsquigarrow$ Bye, bye introspection, hello extra code
- Inconsistent with *other* workarounds
  (to be kept for backward compatibility)
- Can express most, but not all trees
- More, but we're running out of time

Needs our CLI fails to meet

Config files: incomplete & inadequate

Expressive power: weaker than QMP

Single C interface: tedious glue code needed

Introspection: anemic compared to QMP

Part III
Solutions

Actual code to parse -numa

```
Visitor *v;
NumaOptions *numa;

v = opts_visitor_new(opts);
visit_type_NumaOptions(v, NULL, &numa, &err);
visit_free(v);
if (!numa) {
    [error out...]
}
switch (numa->type) {
case NUMA_OPTIONS_TYPE_NODE:
    uint16_t nodeid = numa->u.node.nodeid;
    [more...]
[more cases...]
```

Actual code to parse -numa

```
Visitor *v;
NumaOptions *numa;

v = opts_visitor_new(opts);
visit_type_NumaOptions(v, NULL, &numa, &err);
visit_free(v);
if (!numa) {
    [error out...]
}
switch (numa->type) {
case NUMA_OPTIONS_TYPE_NODE:
    uint16_t nodeid = numa->u.node.nodeid;
    [more...]
[more cases...]
```

QAPI type for -numa's optarg

Actual code to parse -numa

```
Visitor *v;
NumaOptions *numa;

v = opts_visitor_new(opts);
visit_type_NumaOptions(v, NULL, &numa, &err);
visit_free(v);
if (!numa) {
    [error out...]
}
switch (numa->type) {
case NUMA_OPTIONS_TYPE_NODE:
    uint16_t nodeid = numa->u.node.nodeid;
    [more...]
[more cases...]
```

Check and map to QAPI type

Actual code to parse -numa

```
Visitor *v;
NumaOptions *numa;

v = opts_visitor_new(opts);
visit_type_NumaOptions(v, NULL, &numa, &err);
visit_free(v);
if (!numa) {
    [error out...]
}
switch (numa->type) {
case NUMA_OPTIONS_TYPE_NODE:
    uint16_t nodeid = numa->u.node.nodeid;
    [more...]
[more cases...]
```

Look Ma, no strings!

Okay, but what's a QAPI type?

QAPI types are defined in a QAPI schema like this:

```
{ 'union': 'NumaOptions',
  'base': { 'type': 'NumaOptionsType' },
  'discriminator': 'type',
  'data': {
    'node': 'NumaNodeOptions',
    [more variants...] }}
{ 'enum': 'NumaOptionsType',
  'data': [ 'node', [more values...] ] }
```

QAPI compiles them to C types plus code for serializing to/from JSON, introspection, ...

QAPI C type NumaOptions

```
typedef enum NumaOptionsType {
    NUMA_OPTIONS_TYPE_NODE = 0,
    [more type values...]
} NumaOptionsType;

struct NumaOptions {
    NumaOptionsType type;
    union { /* union tag is @type */
        NumaNodeOptions node;
        [more variants...]
    } u;
};
```

Where QAPI beats QemuOpts

- More expressive type system
  - enumerations
  - algebraic types
  - arbitrarily nested
- More precise introspection
- Generated C types beat `QemuOpts` & strings
  - Interfaces made for QMP use nice C types
  - Interfaces made for CLI use `QemuOpts` (and suck)
  - Driving single interface requires conversion

Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

$\rightsquigarrow$ Plan B: QemuOpts mapped to QAPI takes over

Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

$\rightsquigarrow$ Plan B: QemuOpts mapped to QAPI takes over

However:

- Option is then defined in *three* places

Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

$\rightsquigarrow$ Plan B: QemuOpts mapped to QAPI takes over

However:

- Option is then defined in *three* places
- Assumes flat, conflicts with dotted keys

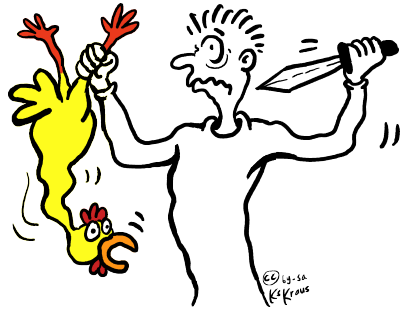
Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

$\rightsquigarrow$ Plan B: QemuOpts mapped to QAPI takes over

However:

- Option is then defined in *three* places
- Assumes flat, conflicts with dotted keys
- Compatibility headaches



Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

$\rightsquigarrow$ Plan B: QemuOpts mapped to QAPI takes over

However:

- Option is then defined in *three* places
- Assumes flat, conflicts with dotted keys
- Compatibility headaches

Fixable, but too much stuff bolted onto QemuOpts

$\rightsquigarrow$ Plan C: QAPI takes over
Status: cooking ☺

QAPIfy CLI option definition

```
##
# @--msg:
# prepend a timestamp to each log message
##
{ 'option': '-msg',
  'data': { '*timestamp': 'bool' },
  'help': [
    "-msg timestamp[=on|off]",
    "      change the format of messages",
    "      on|off controls leading timestamps"] }
```

Observe:

- Option defined in *one* place: QAPI schema
- Like QMP command less 'returns' plus 'help'

QAPI-generated code

All option definitions together compile to:

```
QAPIOption *qapi_options_parse(int argc,
                               char *argv[]);
```

Takes argument vector

Returns array of parsed options `QAPIOption[]`
`QAPIOption` is tagged union of option arguments

New code to parse -numa

```
Visitor *v;
NumaOptions *numa = qopt[i].u.numa;

v = opts_visitor_new(opts);
visit_type_NumaOptions(v, NULL, &numa, &err);
if (!numa) {
    [error out...]
}
switch (numa->type) {
case NUMA_OPTIONS_TYPE_NODE:
    uint16_t nodeid = numa->u.node.nodeid;
    [more...]
[more cases...]
```

QemuOpts middleman cut out

Alternative optarg syntax: JSON

`qapi_options_parse()` supports JSON in addition to dotted *key*=*value*,...

Dotted keys are

- nicer for simple cases
- needed for backward compatibility 

JSON is

- more general
- recommended for programmatic use

Can this meet our CLI needs?

Config files ✓ (JSON)

can read config just like we read QMP

Can this meet our CLI needs?

Config files ✓ (JSON)

can read config just like we read QMP

Expressive power ✓

same as QMP with JSON

close with dotted keys

Can this meet our CLI needs?

Config files ✓ (JSON)

can read config just like we read QMP

Expressive power ✓

same as QMP with JSON

close with dotted keys

Single C interface ✓

types shared with QMP

Can this meet our CLI needs?

Config files ✓ (JSON)

can read config just like we read QMP

Expressive power ✓

same as QMP with JSON

close with dotted keys

Single C interface ✓

types shared with QMP

Introspection ✓

can do just like QMP

Can this meet our CLI needs?

Config files ✓ (JSON)

can read config just like we read QMP

Expressive power ✓

same as QMP with JSON

close with dotted keys

Single C interface ✓

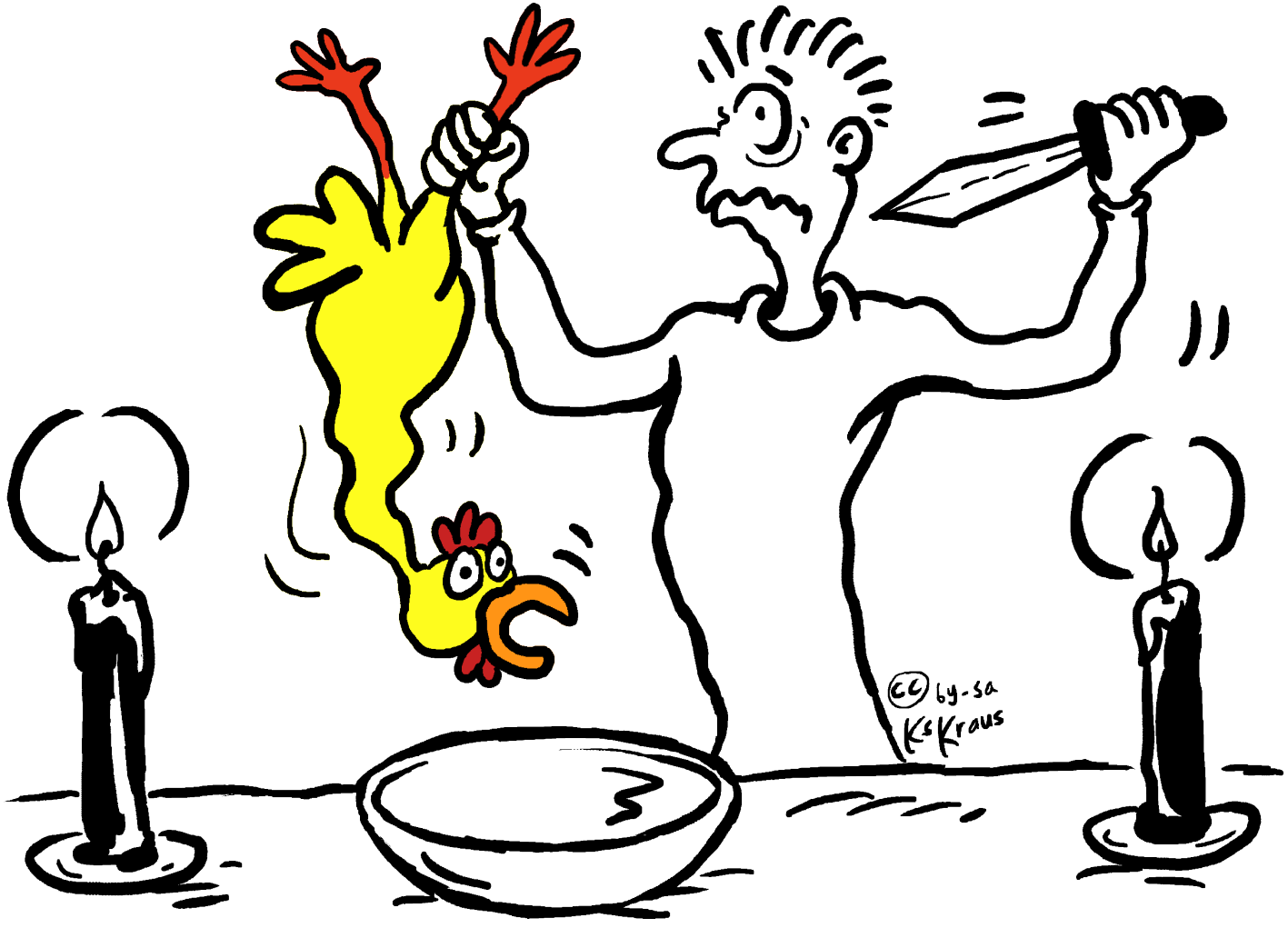
types shared with QMP

Introspection ✓

can do just like QMP

Backward compatibility ?

Backward compatibility



Backward compatibility

- Existing 20+ ad hoc parsers
  - Short term: make optarg a string, pass to parser (bye, bye introspection)
  - Long term: support alternate parsers

Backward compatibility

- Existing 20+ ad hoc parsers
  - Short term: make optarg a string, pass to parser (bye, bye introspection)
  - Long term: support alternate parsers
- Existing bolted-on dotted keys
  - I *think* we're good there

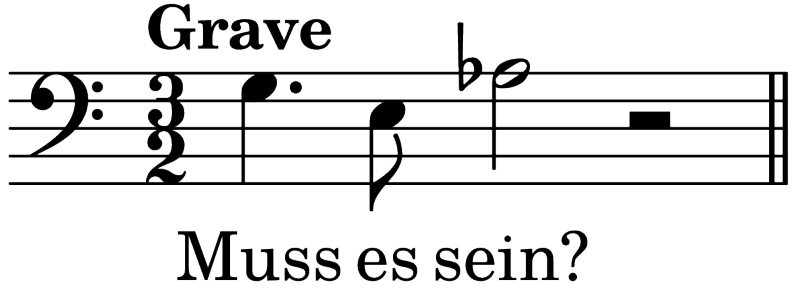
Backward compatibility

- Existing 20+ ad hoc parsers
  - Short term: make optarg a string, pass to parser (bye, bye introspection)
  - Long term: support alternate parsers
- Existing bolted-on dotted keys
  - I *think* we're good there
- Existing bolted-on conversion to QAPI
  - Replicate its extra features and corner cases: flattening, integer lists, ...

Backward compatibility

- Existing 20+ ad hoc parsers
  - Short term: make optarg a string, pass to parser (bye, bye introspection)
  - Long term: support alternate parsers
- Existing bolted-on dotted keys
  - I *think* we're good there
- Existing bolted-on conversion to QAPI
  - Replicate its extra features and corner cases: flattening, integer lists, ...
- QemuOpts eccentricities
  - Replicate those too: syntactic sugar, trickery around repeated keys, special key `id`, ...

Must it be?



Backward compatibility is a choice

We choose how much pain to inflict on ourselves to avoid inconveniencing users

Up to what point are the opportunity costs worth it?

Status: cooking, needs work

Posted: [RFC PATCH] Command line QAPIfication

Highlights and (some of the) lowlights:

- All options QAPIfied
- Most option arguments not fully QAPIfied, yet
  (backward compatibility work hiding here)
- Introspection works
- Config file not yet implemented
- Generated docs look more terrible than usual

Questions?

Thanks & good bye

You might find these links useful:

- [RFC PATCH] Command line QAPIfication
  http://lists.gnu.org/archive/html/qemu-devel/2017-10/msg00209.html
- QEMU interface introspection: from hacks to solutions
  http://www.linux-kvm.org/page/KVM_Forum_2015

No rubber chickens were harmed in the making
of this presentation