advanced go: reflection-based debug console pt. 1

A programming article by Efron Licht

September 2023

more articles

- advanced go & gamedev
    1. advanced go: reflection-based debug console
    2. reflection-based debug console: autocomplete
- go quirks & tricks
    1. declaration, control flow, typesystem
    2. concurrency, unsafe, reflect
    3. arrays, validation, build constraints
- starting software
    1. start fast: booting go programs quickly with `inittrace` and `nonblocking[T]`
    2. docker should be fast, not slow
    3. have you tried turning it on and off again?
    4. test fast: a practical guide to a livable test suite
- faststack: analyzing & optimizing gin's panic stack traces
- simple byte hacking: a uuid adventure

In this article, we'll cover how and why to build a fully-featured debug console that allows live editing of a program's state, such as:

- moving around the player, enemies, or UI elements

ref_move_ui.mp4.gif

- live debug watch window of anything in the game state (& modifying it)

ref_ui_color_modification.mp4.gif

and much more.

1. Motivation

For the last few months I've been working nearly full-time on my own 2D Game, 'Tactical Tapir', a top-down shooter a-la Nuclear Throne.

Some concept art from the excellent Brian Mulligan:

./tt_concept_art.png

./barry_concept_art.png

some in-game art (also by Brian):

./tt_tt.png
./cpt_barry.png

Unlike most games, I'm not using a conventional engine: it's almost all hand-written Go code, using `[ebiten](https://ebiten.org/)` to sit between my code & the GPU and do some input handling. Working with games is a new domain for me: systems are not as isolable as they are in web development or device drivers. In some ways, the questions game development asks are much more subjective: instead of asking "is this code correct?", you ask "does this feel nice?" "Testing" games is very different rather from say, a web server: you have 'squishy', subjective requirements like 'does this feel nice?' or 'is this fun?' rather than 'does this code meet the spec?'. There's no spec for fun, at least as far as I know.

That is, I need to answer questions like this:

- "see if enemies will friendly fire by placing them in each others' line of fire"
- "examine pathfinding by placing enemies in a maze"
- "see how movement feels with different values for speed, acceleration, etc"
- "refill the player's ammo"
- "adjust coloration of fonts for legibility"

To accomplish that, I'll need to be able to quickly and easily:

- spawn an enemy or obstacle
- change player traits (health, movement speed, etc.)
- manipulate the positioning, size, or look of UI elements
- track the state of a variable or group of related variables from frame-to-frame.

This is actually pretty easy in an interpreted language like LUA or Python, but tricky or impossible in a some compiled languages. Luckily, Go's powerful suite of reflection tools makes this possible.

2. An (incredibly brief) introduction to game development

Before we get into the console, let's talk about Game development in general and the process that led to me making a console.

A game is a program that runs in a loop. Each update, or `frame`, does the following:

- take player input
- update the state of the game using that input and the previous state
- draw the relevant part of that state to the screen.

That is, our game loop should look like this:

```
for tick := 0; ; tick++ {
    inputs := input.ThisFrame()
    debugUpdate(game, inputs)
    if err := game.Update(inputs, tick); err != nil {
        log.Fatalf("shutdown: update(): %v")
    }
    if err := game.Draw(screen); err != nil {
        log.Fatalf("shutdown: draw(): %v")
    }
}
```

3: Naive cheats

I started by adding individual keyboard-triggered cheats to the game.

We'll use keyboard inputs to trigger the cheats. We don't want the player to trigger them accidentally, so we'll require that they hold down the `shift` and `ctrl` keys while pressing `H` and `A` respectively.

```
func applyCheats(g *Game, input Inputs) {
    if input.Held[KeyShift] && input.Held[KeyCtrl] {// check for cheats: if no ctrl+shift, no cheats
        if input.JustPressed[KeyA] {
            log.Println("infinite ammo")
            g.Player.Ammo = math.Inf(1)
        }
        if input.JustPressed[KeyH] {
            log.Println("infinite health")
            g.Player.HP = math.Inf(1)
        }
    }
}
```

Which makes our game loop look like this, checking and applying cheats before each update:

```
func (g *Game) Update(input Inputs) {
for tick := 0; ; tick++ {
    inputs := input.ThisFrame()
    applyCheats(game, inputs)
    debugUpdate(game, inputs)
    if err := game.Update(inputs, tick); err != nil {
        log.Fatalf("shutdown: update(): %v")
    }
    if err := game.Draw(screen); err != nil {
        log.Fatalf("shutdown: draw(): %v")
    }
}
}
```

This works pretty well. In fact, we can visualize it as a kind of table:

cheat	key	description
∞ ammo	`ctrl+shift+A`	set `State.Player.Ammo` to math.Inf(1)
∞ hp	`ctrl+shift+H`	set `State.Player.HP` to math.Inf(1)

Which naturally suggests using a map to store the cheats:

```
var cheats map[Key]struct {
    description string
    apply func(*Game)
} {
    KeyA: {
        description: "spawn ammo",
        apply: func(g *Game) {
            g.Pickups = append(g.Pickups, AmmoPickup{...})
        },
    },
    KeyH: {
        description: "spawn health",
        apply: func(g *Game) {
            g.Pickups = append(g.Pickups,
HealthPickup{...})
        },
    },
}
func applyCheats(g *Game, input Inputs, cheats
map[Key]struct {
    description string
    apply func(*Game)
}) {
    if input.Held[KeyShift] && input.Held[KeyCtrl] {//
check for cheats: if no ctrl+shift, no cheats
        for key, cheat := range cheats {
            if input.JustPressed[key]
                log.Println(cheat.description)
                cheat.f(g)
            }
```

```
28          }
29      }
```

While this works well for some tasks, a few limitations are immediately apparent:

- Every new field we touch requires it's own function, even if we have multiple fields that are logically related.
- Those functions must be niladic: that is, they don't take any arguments, they just modify the gamestate directly.
- So for each field, and for each value we want to set it to, we need a separate function.

For some things, like 'infinite HP', this is fine. But for other things, like 'exactly 28 HP', this is absurd. You'd either need to write a function for every possible value, or expand the system somehow to take arguments. We need a more general approach.

4: A more general approach

The default API for general-purpose inputs is the command line. That is, prompt that takes a line of text, parses it as a command, and executes it.

The TacticalTapir console has two layers: the terminal, which takes user input and translates it into lines of text, and the shell, which parses those lines of text into commands and executes them.

That is, every frame, we'll `UpdateTerm()` until the user hits enter, then we'll `ParseCommand()` and `Exec()` that command, updating the gamestate as necessary.

Additionally, every frame, we'll `UpdateState` based off of the current gamestate and preivious `Commands`.

Additionally, we'd like to generate completions for the user as they type - we'll get to that later.

```
1   // A debug console that allows you to inspect and modify the
    game state at runtime.
2   // It's accessed by pressing ~ (shift+`).
3   // Create one with New(), passing in a pointer to the root
    of the struct you want to inspect. Call Update() every frame
    with a pointer to the same value and the current input.Keys
4   // to see if you got a *Command, then Exec() it to modify
    the gamestate.
5   type Console struct {
6       State       // State of the game modified or watched by
    the console.
7       Terminal    // Terminal state: input, history, etc;
    never touches the game state.
8       Completions // Possible completions for the current
    line.
9   }
```

Terminal

A basic terminal includes the following navigational features:

key(s)	description	gif
'a'..'z' or other printable characters	insert the character at the cursor	insert char
← and →	move the cursor left and right	cursor left and right
⇧ + ←, ⇧ + →	move the cursor left and right a word	cursor word
⌫	delete the character to the left of the cursor	bschar
⇧ + ⌫	delete a word to the left of the cursor	bsword
DEL	delete the character to the right of the cursor	delrchar
⇧ + DEL	delete a word to the right of the cursor	delrword
↑ and ↓	navigate the history, if there is one: they don't wrap.	usehist
⇧ + ↑ and ⇧ + ↓	go to the start and end of the history.	usehist
⏎ (enter)	submit the current line of text	**SEE BELOW**
↹ (tab)	autocomplete the current line of text	autocomplete

Working backwards, we'll need to keep track of the following:

- the current line of text
- the cursor position
- the history of previous lines of text
- our last viewed index in the history
- the suggestion for the current line of text, if any.

Additionally, there are a few gotchas we'd like to avoid:

- We should allow players to hold down a navigational or printable key to repeat it, but without causing unintentional inputs; that is, a reasonable amount of 'debouncing'.
- We'd like to avoid allocating memory for each character the player types.

As a Go struct, this looks like:

```
1   // Terminal is the state of the current text input; i.e.,
    the cursor, current line, suggested completion, etc.
2   // Call CurrentLine(), Cursor(), and SuggestedCompletion()
    to get read-only views of the current line, cursor
    position, and suggested completion to
3   // emulate a terminal for it.
```

```
 4  type Terminal struct {
 5      history     History
 6      historySave struct {
 7          time time.Time
 8          prev History
 9          sync.Mutex
10      }
11
12      curLine              string    // current line of
    (normalized) input. View from outside with CurrentLine()
13      curLineRunes         []rune    // runes to append to
    curLine: reusable buffer to avoid  extra allocations
14      cursor               int       // c.cursor position in
    curLine. View from outside with Cursor().
15      lastNavigationRepeat time.Time // time of last
    navigation repeat
16
17      suggestedCompletion string // suggested completion for
    curLine. View from outside with SuggestedCompletion().
18  }
```

Each frame, we'll `UpdateTerm(&term, keys)` with the state of the terminal and the current input, and `DrawTerm` to render it to the screen. (Drawing is outside the scope of this article, though - I hope to get to it soon!)

Key API

Before we get to the implementation, let's quickly go over the API for the `input.Keys` type. The `ebiten.Key` type represents a physical key on a 109-key ANSI keyboard, the most common keyboard layout in the US. Other layouts exist, but we're O.K. with dropping the few bonus keys on, say, the JIS layout.

We'd like an API that looks like this:

```
1  package input // input/input.go
2  type Keys struct { /* some fields */ }
3  // Pressed starting this frame.
4  func (k Keys) Pressed(ebiten.Key) bool {}
5  // Released starting this frame
6  func (k Keys) Released(ebiten.Key) bool {}
7  // Continuously held down; either pressed this frame or
   continuing from a previous frame.
8  func (k Keys) Held(ebiten.Key) bool {}
```

I.e., if on five frames we got the following inputs for the letter 'E':

0, 1, 1, 0, 0

The functions would return the following:

key	held	pressed	released	note
0	F	F	F	ground
1	T	T	F	rising edge
1	T	F	F	continuing
0	F	F	T	falling edge
0	F	F	F	ground

But how best to implement this?

The natural approach is to store the state of each key in a map:

```
type Keys struct { Held, Pressed, Released
map[ebiten.Key]bool}
```

This isn't a bad approach, but maps can be a little fiddly to access concurrently, and every access requires a pointer indirection and a hash lookup. We can do better.

Another approach is simply three arrays of bools:

```
type Keys struct { held, pressed, released
[ebiten.KeyMax+1]bool} // +1 because ebiten.KeyMax is the
highest key, not the number of keys; I think this is a bug
in ebiten.
```

This is straightforward and requires no indirection, using a single byte per key - 327 bytes in total. But we can do better.

Actually, we don't need 324 bits of information, since we can calculate `Held()`, `Pressed()`, and `Released()` by storing the electrical signal (0 or 1) for each key in the current frame and the previous frame and looking for rising and falling edges:

```
type Keys struct { thisFrame, lastFrame
[ebiten.KeyMax+1]bool}
func (k Keys) Pressed(key ebiten.Key) bool {
    return k.thisFrame[key] && !k.lastFrame[key]
}
func (k Keys) Held(key ebiten.Key) bool {
    return k.thisFrame[key]
}
func (k Keys) Released(key ebiten.Key) bool {
    return !k.thisFrame[key] && k.lastFrame[key]
}
```

That knocks us down to 218 bytes, but we can do better. Why store a bool for each key when we can store a single bit? We can use a bitset to store the state of each key by splitting the keys into two groups of 64 bits and do some bit twiddling to access them:

```
struct Keys {thisFrame, lastFrame struct { low, hi
uint64 }}

```

```
 3
 4  func (k Keys) Held(key ebiten.Key) bool {
 5      // read as: "check whether the key'th bit is set in
    thisFrame.low"
 6      if key < 64 {
 7          return k.thisFrame.low & (1 << key) != 0
 8      }
 9      // read as: "check whether the (key-64)th bit is set in
    thisFrame.hi"
10      key -= 64
11      return k.thisFrame.hi & (1 << key) != 0
12  }
13  func (k Keys) Pressed(key ebiten.Key) bool {
14      if key < 64 {
15          return k.thisFrame.low & (1 << key) != 0 &&
    k.lastFrame.low & (1 << key) == 0
16      }
17
18      key -= 64
19      return k.thisFrame.hi & (1 << key) != 0 &&
    k.lastFrame.hi & (1 << key) == 0
20  }
21
22  func (k Keys) Released(key ebiten.Key) bool {
23      if key < 64 {
24          return k.thisFrame.low & (1 << key) == 0 &&
    k.lastFrame.low & (1 << key) != 0
25      }
26      key -= 64
27      return k.thisFrame.hi & (1 << key) == 0 &&
    k.lastFrame.hi & (1 << key) != 0
28  }
```

With this design, we can store the state of every key on the keyboard in only 32 bytes (or four 64-bit words). This makes the input state small enough to cheaply copy wherever we need it. Additionally, if we ever needed to access it concurrently, we could use atomic operations to do so without any locks! Additionally, this opens up the possibility of using bit operations to enable or disable 'groups' of keys at once in very few instructions.

OK, that's enough about the input API. Let's get back to updating the terminal.

`UpdateTerm()`

The following is an annotated version of the core of `UpdateTerm()`, stripping out some inessential features like clipboard support and saving the history to disk.

```
 1  // Update the terminal state with the given input. If the
    user pressed enter, parse the current line and return a
    *Command.
 2  //
```

```
 3  // # cursor:
 4  //
 5  // ← and → move the cursor left and right
 6  //
 7  // ⇧ + ← and ⇧ + → move the cursor left and right a word
 8  //
 9  // # deletion
10  //
11  // ⌫ deletes the character to the left of the cursor
12  //
13  // ⇧ + ⌫ deletes a word to the left of the cursor
14  //
15  // DEL deletes the character to the right of the cursor
16  //
17  // ⇧ + DEL deletes a word to the right of the cursor
18  //
19  // # history:
20  //
21  // ↑ and ↓ navigate the history, if there is one: they
    don't wrap.
22  //
23  // ⇧ + ↑ and ⇧ + ↓ go to the start and end of the history.
24  func UpdateTerm(t *Terminal, keys input.Keys) (cmd
    *Command, err error) {
25      // bounds check the cursor & suggested completion,
    just in case
26      // we forgot to clean up after ourselves last frame
27
28      if !strings.HasPrefix(t.suggestedCompletion,
    t.curLine) {
29          t.suggestedCompletion = ""
30      }
31      t.cursor = util.Clamp(t.cursor, 0, len(t.curLine))
32
33      // how many navigation keys are pressed? if more than
    one, we do nothing rather than guess.
34
35      var (
36          up, down, left, right, backspace, del bool
37          shift                                 =
    keys.Held(ebiten.KeyShift)
38          tab, enter                            =
    keys.Pressed(ebiten.KeyTab), keys.Pressed(ebiten.KeyEnter)
39      )
40      if time.Since(t.lastNavigationRepeat) >
    time.Millisecond*200 {
41          up = keys.Pressed(ebiten.KeyUp) ||
    keys.Held(ebiten.KeyUp)
42          down = keys.Pressed(ebiten.KeyDown) ||
    keys.Held(ebiten.KeyDown)
43          left = keys.Pressed(ebiten.KeyLeft) ||
    keys.Held(ebiten.KeyLeft)
```

```
44          right = keys.Pressed(ebiten.KeyRight) ||
    keys.Held(ebiten.KeyRight)
45          backspace = keys.Pressed(ebiten.KeyBackspace) ||
    keys.Held(ebiten.KeyBackspace)
46          del = keys.Pressed(ebiten.KeyDelete) ||
    keys.Held(ebiten.KeyDelete)
47      } else {
48          up = keys.Pressed(ebiten.KeyUp)
49          down = keys.Pressed(ebiten.KeyDown)
50          left = keys.Pressed(ebiten.KeyLeft)
51          right = keys.Pressed(ebiten.KeyRight)
52          backspace = keys.Pressed(ebiten.KeyBackspace)
53          del = keys.Pressed(ebiten.KeyDelete)
54      }
55      // how many navigation keys are pressed? if more than
    one, we do nothing rather than guess.
56
57      /* -- design note --
58          this is the most important part of this function,
    since it massively reduces the number of cases we need to
    handle. rather than worrying about what to do if the user
    presses both left and right, or if they both insert a
    character and delete a character, we just ignore those
    cases, assuming they're user error or bouncing keys.
59          the original design of this function was hundreds
    of lines longer and distinctly buggier. This is the fourth
    revision.
60      */
61
62      b2i := func(b bool) int { if b { return 1 }; return
    0 }
63      switch count := b2i(up) + b2i(down) + b2i(left) +
    b2i(right) + b2i(backspace) + b2i(del) + b2i(tab) +
    b2i(enter); count {
64      default: // more than one key pressed: do nothing
65          return nil, nil
66      case 0: // no navigation keys pressed: insert
    characters & adjust cursor position
67          t.curLineRunes =
    ebiten.AppendInputChars(t.curLineRunes[:0])
68          t.curLine = normalize(t.curLine[:t.cursor] +
    string(t.curLineRunes) + t.curLine[t.cursor:])
69
70          // adjust the cursor position for the new
    character(s)
71          t.cursor += len(t.curLineRunes)
72          t.cursor = max(t.cursor, 0)
73          t.cursor = min(t.cursor, len(t.curLine))
74          return
75      case 1:
76          t.lastNavigationRepeat = time.Now()
77      }
78      /* -- design note --
```

```
79          note that we check shift and also !shift. this is
    technically redundant, but it protects us against ordering
    bugs, since it means the switch cases are guaranteed to be
    mutually exclusive.
80      */
81      switch {
82      default: // no navigation to do, like backspace on an
    empty line
83          return nil, nil
84      case enter: // execute the current line
85          line := t.curLine
86          t.curLine, t.cursor = "", 0 // reset the current
    line
87          c, err := ParseCommand(strings.Fields(line))
88          if err != nil {
89              return nil, err
90          }
91          // it's a valid command (in form, at least): add
    it to the history
92          const maxHistLen = 64
93          if len(t.history.Lines) >= maxHistLen { // trim
    the history:
94              // copy the back half of the history to the
    front
95              copy(t.history.Lines[:len(t.history.Lines)/
    2], t.history.Lines[len(t.history.Lines)/2:])
96              // then truncate the back half
97              t.history.Lines =
    t.history.Lines[:len(t.history.Lines)/2]
98          }
99          // now there's definitely room in the history
100         t.history.Index = len(t.history.Lines) - 1
101         t.history.Lines = append(t.history.Lines, line)
102
103         return &c, nil
104     case tab && t.suggestedCompletion != "": //
    autocomplete the current line
105         t.curLine = t.suggestedCompletion
106         t.cursor = len(t.curLine) // move the cursor to
    the end of the line
107         return nil, nil
108     case left && shift: // move cursor left a word
109         t.cursor =
    max(strings.LastIndexAny(t.curLine[:t.cursor], " \n"), 0)
110         return nil, nil
111     case right && shift: // move cursor right a word
112         if i := strings.IndexAny(t.curLine[t.cursor:], "
    \n"); i >= 0 {
113             t.cursor += i + 1
114         } else {
115             t.cursor = len(t.curLine)
116         }
117         return nil, nil
118     case left && !shift: // move cursor left.
```

```
119             t.cursor = max(t.cursor-1, 0)
120             return nil, nil
121         case right && !shift: // move cursor right
122             t.cursor = min(t.cursor+1, len(t.curLine))
123             return nil, nil
124         // --- deletion ---
125         case backspace && shift && t.cursor > 0: // delete
    word to the left
126             i :=
    max(strings.LastIndexAny(t.curLine[:t.cursor], " \n"), 0)
127             t.curLine = t.curLine[:i] + t.curLine[t.cursor:]
128             t.cursor = i
129             return nil, nil
130         case backspace && !shift && t.cursor > 0: // delete
    char to the left
131             if t.cursor > 0 {
132                 t.curLine = t.curLine[:t.cursor-1] +
    t.curLine[t.cursor:]
133                 t.cursor--
134             }
135             return nil, nil
136         case del && shift && t.cursor < len(t.curLine): //
    delete word to the right
137             i := strings.IndexAny(t.curLine[t.cursor:], " \n")
138             if i < 0 {
139                 i = len(t.curLine)
140             } else {
141                 i += t.cursor
142             }
143             t.curLine = t.curLine[:t.cursor] + t.curLine[i:]
144             return nil, nil
145
146         case del && !shift && t.cursor < len(t.curLine): //
    delete char to the right
147             t.curLine = t.curLine[:t.cursor] +
    t.curLine[t.cursor+1:]
148             return nil, nil
149
150         // --- history navigation ---
151         case down && shift: // goto end of history
152             t.history.Index = len(t.history.Lines) - 1
153             t.curLine, t.cursor =
    t.history.Lines[t.history.Index],
    len(t.history.Lines[t.history.Index])
154             return nil, nil
155
156         case up && shift: // goto start of history
157             t.history.Index = 0
158             t.curLine, t.cursor =
    t.history.Lines[t.history.Index],
    len(t.history.Lines[t.history.Index])
159             return nil, nil
160         case up && !shift: // prev line in history
161             if t.history.Index--; t.history.Index < 0 {
```

```
162                 t.history.Index = 0
163             }
164             t.curLine, t.cursor =
      t.history.Lines[t.history.Index],
      len(t.history.Lines[t.history.Index])
165             return nil, nil
166         case down && !shift: // next line in history
167             if t.history.Index++; t.history.Index >=
      len(t.history.Lines) {
168                 t.history.Index = len(t.history.Lines) - 1
169             }
170             t.curLine, t.cursor =
      t.history.Lines[t.history.Index],
      len(t.history.Lines[t.history.Index])
171             return nil, nil
172         }
173         // unreachable
174     }
```

ParseCommand

Our commands will have an `OPCODE` and zero or more `ARGS`. `ARGS` will be literals representing strings or numbers (e.g. `100`, `hello`), `PATH`s to fields of the game state, or `AUGOP`s (augmented assignment operators) like `+=` or `*=`.

`PATH`s will only be able to reach exported (i.e., capitalized) fields of the game state, respecting go's usual rules for visibility. In order to make this easier to work with, all commands will be case-insensitive; in fact, we'll lowercase them before parsing. This does allow for some ambiguities that could lead to 'unreachable' fields. If we had `struct S {XY, xy int}`, then `s.xy` would be ambiguous.

I resolve this problem by *not making structs like that*, but it's worth noting this limitation.

ITEM	description	example
`OPCODE`	specifies a console command	`watch`, `mod`, `cpin`
`PATH`	a path to a field of the game state.	`player.hp`, `player.pos.x`
`AUGOP`	an augmented assignment operator.	`+=`, `*=`, `%=`
`LITERAL`	a literal value, interpreted by the OP	`100`, `hello`

Commands will be of the form: `OP arg1 arg2 ... argN`, but we'll also allow the form `arg1 AUGOP arg2`, which is equivalent to `mod arg1 AUGOP arg2`. (That is, `mod` is implied as the opcode.)

Some example commands:

op	args	description
`watch`	`watch player.hp`	add the player's hp to the debug watch window

op	args	description
`cpin`	`cpin player.hp 100`	“reset the player’s HP to 100 every frame, “pinning it” to a constant value.
`mod`	`mod player.hp = 100`	set the player’s hp to 100
`mod`	`mod player.hp += 100`	add 100 to the player’s hp
`mod`	`player.x /= 2`	halve the player’s x position; ‘mod’ is implied

We’ll represent operators as an enum:

```
type OpCode
int16 // operator. hard to imagine needing more than 256 operators, but we'll use 16 bits just in case.
const (
    // Unknown or invalid operator
    Unknown OpCode = iota
    // WATCH adds a field to the debug watch window. Grammar: `watch <path>`
    WATCH
    // CPIN "pins" a field to a constant value, resetting it every frame. Grammar: `cpin <path> <path_or_literal>`
    CPIN
    // MOD modifies a field. Grammar: `mod <path> <augop> <path_or_literal>`, or <path> <augop> <path_or_literal>
    MOD
)
```

Before we get to parsing, we have an important design decision to make: do we represent commands as a single struct regardless of opcode, or as a separate struct for each command?

A **single struct** would look like this:

```

type Command struct {
    Op Opcode
    Args []string
}
func Exec[T any](pt *T, Command) (string, error) {
    switch c.Op {
    default:
        return "", fmt.Errorf("unknown opcode %d", c.Op)
    case Watch:
    case Cpin:
    case Mod:
    }
}
```

Whereas a **separate struct for each command** would look like this:

```
```

```
type Command interface {
    Opcode() Op
    Exec(reflect.Value) (string, error)
}
type Mod struct {
    LHS, AugOp string
    RHS reflect.Value
}
type Watch struct {
    Path string
}
type Cpin struct {
    Path string
    Value reflect.Value
}
func (c Mod) Opcode() Op { return Mod }
func (c Mod) Exec(v reflect.Value) (string, error) {
    // ...
}
func (c Watch) Opcode() Op { return Watch }
func (c Watch) Exec(v reflect.Value) (string, error) {
    // ...
}
func (c Cpin) Opcode() Op { return Cpin }
func (c Cpin) Exec(v reflect.Value) (string, error) {
    // ...
}

```

- A single struct requires less code, but is 'stringly typed': we can't use the type system to enforce that the arguments to `mod` are a path, an augop, and a path or literal. We'll just have to make sure that `Exec()` does the right thing. As our list of commands grows, this may become more difficult to maintain, since we'll have a "god switch" in `Exec()` that handles all commands.

- A separate struct for each command requires more code, but allows us to add new commands without touching the existing code. Additionally, once we have a parsed command, we have stronger guarantees about what it contains: we know that a `Mod` command has a `Path`, an `AugOp`, and a `Value`, and we can use the type system to enforce that.

- A third, best option exists, but we can't use it in Go. Languages with sum types (sometimes called "tagged unions" or "enums") could express a `Command` like this:

```
    // this is rust code: don't worry about it too much.
    enum AugOp { Add, Sub, Mul, Div, Mod, Pow, BitAnd, BitOr, BitXor, BitClear, Shl, Shr }
    enum Command {
        Set(Path, Value),
        Watch(Path),
        Cpin(Path, Value),
        AugAssign(path: Path, op: AugOp, value: Value),
```

```
8     }
```

This would combine the best of both worlds: we'd have a single type to represent all commands, but we could guarantee the proper structure of each.

Go doesn't have sum types, so **I chose the single struct approach to minimize the total amount of code**.

Because we're not using separate structs, I don't do much validation in ParseCommand: instead, I'll guarantee the bounds when we actually try and execute.

```
1   func ParseCommand(fields []string) (Command, error) {
2
3      switch op := strings.ToLower(fields[0]); op {
4       default: // unknown opcode
5         // is this an augmented assignment operator?
6         if len(fields) == 3 && augassignops[fields[1]] !=
    nil {
7              return Command{MOD, fields[1:]}, nil
8         }
9         return Command{}, fmt.Errorf("unknown operation %q:
    expected one of %v", op, opNames)
10      case "mod":
11        if len(fields) != 4 {
12           return Command{}, fmt.Errorf("mod: expected 3
    args")
13        }
14        return Command{MOD, fields[1:]}
15      case "cpin": // PIN a field to a Constant value
16        if len(fields) != 3 {
17           return Command{}, fmt.Errorf("cpin expects 2
    arguments, got %d", len(fields)-1)
18        }
19        return Command{Op: CPIN, Key: fields[1], Vals:
    fields[2:]}, nil
20      /* other cases omitted; they're pretty similar. */
21     }
22  }
```

OK, easy enough. Now we get to the hard part: how do we implement `Command.Exec()`? That is, *how do we access and modify arbitrary fields of arbitrary structs at runtime?*

4: Reflection

The reflect package allows you to operate on Go values of arbitrary type without knowing what type or types they are ahead of time. Reflect is too big of a subject to cover in detail here. Instead, I'll first show a few examples of what you can do with it, then present a cheatsheet of the most useful types and functions for you to refer to, then we'll get back to the console. If you're completely lost, I recommend reading the

reflect package docs and the reflect tutorial first. Chapter 12 of The Go Programming Language by Kernighan & Donovan is also an excellent resource: see the source code for that chapter's examples here.

examples

First, a few examples to get the idea across.

- #### Get the value of struct fields by name, regardless of type

```
  // https://go.dev/play/p/gh7TMf2-JlE

  var f64type = reflect.TypeOf(0.0)
  //  get the value of "`X`" and "`Y`" fields of a struct, regardless of what type the struct is, as long as they're both _any_ numeric type, even if X or Y are embedded in another struct.
  func getXY(v reflect.Value) (x, y float64, ok bool) {
      if v.Type().Kind() != reflect.Struct { // make sure we have a struct
          return 0, 0, false
      }
      // check if v.X or v.Y would be valid expressions at compile time on the type of v
      vx, vy := v.FieldByName("X"), v.FieldByName("Y")

      if !vx.IsValid() || !vy.IsValid() {
          // they're not, so we can't do it at runtime either
          return 0, 0, false
      }
      // and that f64(v.X) and f64(v.Y) would be valid conversions at compile time
      if !vx.CanConvert(f64type) || !vy.CanConvert(f64type) {
          // they're not, so we can't do it at runtime either
          return 0, 0, false

      }
      // they are: convert them to float64s and return them
      x, y = vx.Convert(f64type).Float(), vy.Convert(f64type).Float()
      return x, y, true
  }
```

IN:

```
  func main() { // https://go.dev/play/p/IiMldZgkEum
      for _, v := range []any{
          &image.Point{1, 2}, // X and Y are `int` in this package!
```

```
            &struct{ X, Y float64 }{3, 4},
            &struct{ image.Point }{image.Point{5, 6}}, // embedded fields
        } {
            v := reflect.ValueOf(v).Elem() // get the Value of the pointer
            x, y, _ := getXY(v)            // get the value of the X and Y fields as float64s
            fmt.Printf("%s: %v, %v\n", v.Type(), x, y)
        }
    }
```

OUT:

```
image.Point: 1, 2
struct { X float64; Y float64 }: 3, 4
struct { image.Point }: 5, 6
```

- #### Set the value of struct fields by name, regardless of type

```
    // https://go.dev/play/p/gh7TMf2-JlE
    var f64type = reflect.TypeOf(0.0)

    // set the value of the "`X`" and "`Y`" fields of a struct, so long as X and Y are both _any_ numeric type, even if X or Y are embedded in another struct.
    // we could use this to, for example, set the position of an object in a game to the position of the mouse cursor.
    func setXY(v reflect.Value, x, y float64) bool {
        if v.Type().Kind() != reflect.Struct {
            return false // not a struct
        }
        vx, vy := v.FieldByName("X"), v.FieldByName("Y")
        if !vx.IsValid() || !vy.IsValid() {
            return false // no X or Y field
        }
        if !vx.CanSet() || !vy.CanSet() {
            return false // X or Y is unexported, part of an unexported struct, or isn't in an addressable struct
        }
        if !f64type.ConvertibleTo(vx.Type()) || !f64type.ConvertibleTo(vy.Type()) {
        return false
        }
        vx.SetFloat(x)
        vy.SetFloat(y)
    }
```

IN

```
```go
// https://go.dev/play/p/gh7TMf2-JlE
```
```

```
func main(){
    for _, v := range []any{
        &image.Point{1, 2}, // X and Y are `int` in this package!
        &struct{ X, Y float64 }{3, 4},
        &struct{ image.Point }{image.Point{5, 6}},
    } {
        v := reflect.ValueOf(v).Elem()                // get the Value of the pointer
        x, y, _ := getXY(v)                           // get the value of the X and Y fields as float64s
        fmt.Printf("%s: %v", v.Type(), v.Interface()) // print the type and the values
        setXY(v, x*10, y*10)                          // set the value of the X and Y fields to 10x their original value
        fmt.Printf("-> %v\n", v.Interface())          // print the type and the values
    }
}
```

OUT:

```text
image.Point: (1,2)-> (10,20)
struct { X float64; Y float64 }: {3 4}-> {30 40}
struct { image.Point }: (5,6)-> (50,60)
```

- #### zero out any field of any struct

```
  // zero out the given field of a struct, regardless of the type of struct or field, or whether the field is embedded in another struct.
  func zeroField(v reflect.Value, fieldName string) bool {
  if v.Type().Kind() != reflect.Struct {
          return false // not a struct
      }
      f := v.FieldByName(fieldName)
      if !f.IsValid() {
          return false // no field
      }
      if !f.CanSet() {
          return false // field is unexported, part of an unexported struct, or isn't in an addressable struct
      }
      f.Set(reflect.Zero(f.Type()))
      return true
  }
```

IN:

```
  // https://go.dev/play/p/YO8LmQqqZuJ
```

```
 2     func main() {
 3     type A struct{ F string }
 4     var a = A{"foo"}
 5     fmt.Printf("a: before: %+v\n", a)
 6     zeroField(reflect.ValueOf(&a).Elem(), "F")
 7     fmt.Printf("a: after: %+v\n", a)
 8
 9     type B struct{ F int }
10     var b = B{2}
11     fmt.Printf("b: before: %+v\n", b)
12     zeroField(reflect.ValueOf(&b).Elem(), "F")
13     fmt.Printf("b: after: %+v\n", b)
14
15     }
```

OUT:

```
1     a: before: {F:foo}
2     a: after: {F:}
3     b: before: {F:2}
4     b: after: {F:0}
```

[reflect: types and values]

Reflect operates on three main types: `reflect.Type`, `reflect.Value`, and `reflect.Kind`. `reflect.Type` represents a type, `reflect.Value` represents a value of that type, and `reflect.Kind` represents the underlying primitive type of a `reflect.Type`; that is, something like `int`, `string`, `struct`, `map`, `slice`, etc.

Get a Value from a normal variable via `reflect.ValueOf(t)`, then modify it with the various functions on `reflect.Value`. Pretty much anything you can do in 'ordinary' go you can do with some combination of `reflect.Value`'s methods. E.g., the following snippets are functionally equivalent:

```
1   var n int
2   reflect.ValueOf(&n).Elem().SetInt(50)
```

```
1   func main() {var n int; *(&n) = 50}
```

Or to show it another way:

```
1   reflect.ValueOf(&n). // &
2   Elem(). // *
3   SetInt(50) // =
```

Note the pointers. Since `reflect.ValueOf` is an ordinary function, you'll need to **pass a pointer** if you want to modify one of the arguments, just like any other function.

Find out information about a type via `reflect.TypeOf(t)` or the underlying primitive type via `Type.Kind()`.

In the following notation, eleme `t` is a `reflect.Type`, `v` is a `reflect.Value`, `T` and `B` is are types, and `t` and `b` are values of those types (not `reflect.Values`, but the normal type you get via `':='`, `'var'`, etc.

reflect: cheatsheet

Here's a quick cheatsheet of the types and functions we'll use in this article. Feel free to skip this for now, and come back to it when or if you need it.

types

shorthand	type	obtained via
v	`reflect.Value`	`reflect.ValueOf("some string")`
t	`reflect.Type`	`v.Type()` or `reflect.TypeOf("another string")`
k	`reflect.Kind`	`t.Kind()`
f	`reflect.StructField`	`t.Field()` or `t.FieldByName()` or `t.FieldByNameFunc()`
n	`int8..=int64` or `int`	`n := 2`
b	`bool`	`b := true`
s	`string` or `struct`	`s := "some string"`,`s := struct{foo int}{"foo}`
m	`map`	`m := map[string]int{"a": 1}`
a	`slice` or `array`	`a := []int{1, 2, 3}`

functions

function	description	example	analogous to
`ValueOf`	get a `Value` from an ordinary value	`reflect.ValueOf(int(2))`	
`TypeOf`	get a `Type` from the value	`t := reflect.TypeOf(int(2))`	
Type.Kind	get the underlying primitive type	`t.Kind()`	
—	—	—	

Type.ConvertibleTo	can the type be converted to a different type?	`t.ConvertibleTo(reflect.TypeOf(0))`
Value.Addr	get the address of a value	`v.Addr()`
Value.CanAddr	can the value be addressed?	`v.CanAddr()`
Value.CanConvert	can the value be converted to a different type?	`v.CanConvert(reflect.TypeOf(0))`
Value.Convert	convert a value to a different type	`reflect.ValueOf(&t).Elem().Convert(reflect.`
Value.Elem	dereference a pointer or interface	`v.Elem()`
Value.Field	get the `nth` field of a struct	`v.Field(0)`
Value.FieldByName	for `struct` kinds, get the field with the given name	`v.FieldByName("someField")`
Value.FieldByNameFunc	for `struct` kinds, get the field with the given name, matching the given predicate	`v.FieldByNameFunc(func(s string) bool { ret strings.EqualFold(s, "somefield") })`
Value.Index	for `array` and `slice` kinds, get	`v.Index(0)`

	the `nth` element	
`Value.Interface`	get an ordinary value back from a `Value` (as `any`)	`reflect.ValueOf(2).Interface().(int)`
`Value.Len`	for `array`, `map`, and `slice` kinds, get the length	`v.Len()`
`Value.MapIndex`	for `map` kinds, get the value associated with the given key	`v.MapIndex(reflect.ValueOf("someKey"))`
`Value.Set`	set lhs to rhs, if they're the same `Type`	`v.Set(reflect.ValueOf(2))`

OK, that covers what we'll need for now. Let's get back to the console.

5: Executing commands

In order to execute commands, we'll need to be able to:

- resolve paths to fields of structs, indices of slices or arrays, and values of maps.
- convert values to the correct type
- set the value of a field, index, or map key to a new value.

5.1: Resolving paths

We'd like a function which allows us to access any of the subfields of the Game struct, regardless of how deeply nested they are.

That is, we'd like a function like this:

```
1  // FollowPath follows a path of '.'-separated keys through a
   struct, map, or slice, returning the value at the end of the
   path.
2  // pointers and interfaces will be continually dereferenced.
```

```
3   func ResolvePath[T any](pt *T, path string) (reflect.Value,
    error) {}
4
5   type Inner struct{ X int }
6   type Outer struct{ Inner Inner }
7   type S struct{ Outer Outer }
8   var s = S{Outer: Outer{Inner: Inner{X: 1}}}
9   ResolvePath(&s, "outer.inner.x") // like
    reflect.ValueOf(&s.outer.inner.x).Elem()
```

Additionally, we'd like a single uniform syntax that allows us to access fields of structs, indices of slices or arrays, and values of maps. Taking a cue from `lua`, we'll use `.` as our access operator. All of these should work:

```
 1      type S struct{ N int, A []map[string]int }
 2      func printResolved[T any](pt *T, path string) {
 3          v, err := ResolvePath(pt, path)
 4          if err != nil {
 5              panic(err)
 6          }
 7          fmt.Printf("%s: %v\n", path, v.Interface())
 8      }
 9      func main() {
10          s = S{N: 1, A: []map[string]int{{"a": 1}, {"1",
    3}}}
11          printResolved(&s, "n") // 1
12          printResolved(&s, "a.0") // map[string]int{"0": 3}
13          printResolved(&s, "a.0.a") // 1
14          printResolved(&s, "a.1.0") // 3: note that 0 is
    treated as a string here, since that's the type of the keys
    of the map.
15          printResolved(&s, "a.1.-1") // 3: negative indices
    for slices are treated as python or FORTRAN-style negative
    indices, where -1 is the last element, -2 is the second-to-
    last, etc.
16      }
```

Let's walk through our implementation of `ResolvePath` step by step. First, we'll need a few helper functions: **derefVal** to dereference pointers and interfaces, and **normalize** to unify our path syntax.

```
1   // continually dereference pointers and interfaces until we
    get a non-pointer, non-interface value.
2   // panic if we dereference more than 32 times, since this
    means we've hit some kind of self-referential loop.
3   func derefVal(v reflect.Value) reflect.Value {
4       for i := 0; v.Kind() == reflect.Ptr || v.Kind() ==
    reflect.Interface; i++ {
5           v = v.Elem()
6           if i > 32 {
7               panic("dereferenced 32 pointers, but still got
    a pointer or interface")
8           }
```

```
 9       }
10       return v
11   }
12
13   // whitespaceUnifier replaces all whitespace with a single
     space.
14   var whitespaceUnifier = strings.NewReplacer("\t", " ",
     "\n", " ", "\r", " ")
15
16   // normalize normalizes a string by
17   //  - lowercasing it
18   //  - replacing all whitespace with a single space
19   //  - removing leading and trailing whitespace
20   // warning: given the implementation of normalize, we will
     not be able to access some string map keys.
21   // this may not be appropriate for your use case. Again, I
     solve this problem by "not doing that".
22   func normalize(s string) string {
23       s = strings.ToLower(s)
24       s = strings.TrimSpace(s)
25       old := s
26       for {
27           s = whitespaceUnifier.Replace(s)
28           if s == old {
29               return s
30           }
31           old = s
32       }
33   }
```

Our outside-facing API will take the `.`-separated path:

```
1   func func ResolvePath[T any](pt *T, path string)
    (reflect.Value, error) {
2       return resolvePath(reflect.ValueOf(pt).Elem(),
    strings.Split(path, "."))
3   }
```

And our implementation will step through the path, following each key in turn:

```
 1   // follow a path of key, case-insensitively, through a
     struct, map, or slice, returning the value at the end of
     the path.
 2   // pointers and interfaces will be continually
     dereferenced.
 3   //
 4   //        type S struct{ F struct { A [3]int } }
 5   //        s := S{F: struct{ [3]int }{[3]int{42, 43, 44}}}
 6   //        v, _ := followPath(reflect.ValueOf(s), "f", "a",
     "0")
 7   //        fmt.Println(v)
 8   //     Output: 42
 9   func followPath(root reflect.Value, keys ...string)
     (reflect.Value, error) {
```

```
10      v := derefVal(root) // will be updated once per loop iteration
11
12      for i, field := range keys { // follow the path: e.g., player.pos.x
13          t := v.Type()
14          switch k := t.Kind(); k {
15          default:
16              return v, fmt.Errorf("%s: %v is not a struct, map, slice, or array", root.Type(), strings.Join(keys[:i+1], "."))
17
18          // structs are simple: just get the field by (normalized) name.
19          case reflect.Struct:
20              v = v.FieldByNameFunc(func(s string) bool { return normalize(s) == field })
21              if !v.IsValid() { // field not found
22                  return v, fmt.Errorf("%s: %v has no field %q", root.Type(), strings.Join(keys[:i+1], "."), field)
23              }
24
25          case reflect.Slice, reflect.Array:
26              // treat the key as an integer index. we use strconv.ParseInt to allow users to use hex, binary, or octal indices if they'd like.
27              j64, err := strconv.ParseInt(field, 0, 0)
28              j := int(j64)
29              if err != nil {
30                  return v, fmt.Errorf("%s: %v is not a valid index", root.Type(), strings.Join(keys[:i+1], "."))
31              }
32              if j < 0 {
33                  j += v.Len() // python-style negative indexing; -1 is the last element, -2 is the second-to-last, etc.
34                  // but don't allow silly things like -1000 if the slice only has 3 elements.
35              }
36              if j < 0 || j >= v.Len() {
37                  return v, fmt.Errorf("%s: index %d out of bounds", root.Type(), j) // out of bounds
38              }
39              v = v.Index(j)
40          }
41          case reflect.Map:
42              // treat the key as a map key.
43              // separate branches for uints, ints, floats, strings.
44              // all other types are not supported.
45
46              var key any
47              var err error
```

```
48              // treat this field as a map key, parsing it as
    losslessly as possible into the highest-precision numeric
    Kind we can.
49              // i.e., uint8..64 => uint64, int8..64 =>
    int64, float32 => float64, string => string
50              switch t.Key().Kind() {
51              case reflect.String:
52                  key, err = field, nil
53              case reflect.Int, reflect.Int8, reflect.Int16,
    reflect.Int32, reflect.Int64:
54                  key, err = strconv.ParseInt(field, 0, 64)
55              case reflect.Uint, reflect.Uint8,
    reflect.Uint16, reflect.Uint32, reflect.Uint64,
    reflect.Uintptr:
56                  key, err = strconv.ParseUint(field, 0, 64)
57              case reflect.Float32, reflect.Float64:
58                  key, err = strconv.ParseFloat(field, 64)
59              default:
60                  err =
    fmt.Errorf("%s: %v is not a supported map key type",
    root.Type(), strings.Join(keys[:i+1], "."))
61              }
62              if err != nil {
63                  return v,
    fmt.Errorf("%s: %v is not a valid map key: failed to parse
    as %v: %w", root.Type(), strings.Join(keys[:i+1], "."),
    t.Key().Kind(), err)
64              }
65              // convert the key to actual type of the map's
    key and use it to index the map.
66              // this handles both precision (e.g., uint64 ->
    uint8) and custom types (e.g., type Celsius float64 ->
    float64).
67              // more about this in the next section.
68              key = reflect.ValueOf(key).Convert(t.Key())
69
70              // now index the map
71              v = v.MapIndex(key)
72      }
73      return derefVal(v), nil
74  }
```

5.2: Literals

We'd like all of the following commands to work, without worrying about the type of the fields or the values:
they should "just work":

- `set player.hp 100`
- `set player.hp 100.0`
- `set player.hp player.x`

- `set player.pos npcs.0.pos`

Additionally, we'd like the ability to handle custom types, like colors

converting literals

How to handle literals depends on the type of the field we're setting.

- **strings** require no processing.
- **numbers** can be handled like map keys above: parsing as the highest-precision numeric Kind we can, then converting to the actual type of the field, truncating if necessary.
- **bools** can be parsed using `strconv.ParseBool`.
- **other types** can use the `encoding.TextUnmarshaler` interface, which is implemented by many types in the standard library, including `*time.Time` and `net.IP`. A note here: most of the time, these types require a *pointer* for the method, so we might occasionally need to add a level of indirection to satisfy the interface. **This will have the highest priority**. While it is possible for a type to implement `encoding.TextUnmarshaler` without a pointer receiver (some maps, for example), we will *omit this case*. After all, this console doesn't need to solve *all* problems, just the problems I have.

Additionally, we'd like to handle one last case - the `color.RGBA` struct is already in use throughout the codebase. I could surround it with a wrapper that implements `encoding.TextUnmarshaler` to make the syntax uniform, or I can special-case an exception. Here, I choose to special-case the exception. I find myself doing this more than once, though, I might consider adding wrappers rather than making the code too complicated.

Let's see what this looks like in code:

```
 1  // https://go.dev/play/p/KzqjgzF1PhP
 2
 3  // SetString interprets src as a string literal, and
    attempts to set dst to that value.
 4  // Conversions happen in this order:
 5  // If dst, &dst, *dst, **dst, etc implement
    encoding.TextUnmarshaler, use UnmarshalText([]byte(src))
 6  // Otherwise, if dst is a string, set it to src.
 7  // Otherwise, if dst is a bool, set it to the result of
    strconv.ParseBool(src)
 8  // Otherwise, if dst is a numeric type, set it to the
    result of strconv.ParseFloat(src, 64).
 9  func SetString(dst reflect.Value, src string) error {
10      // special cases: do dst, &dst, *dst, **dst, etc
    implement encoding.TextUnmarshaler?
11      if dst.CanAddr() {
12          if x, ok := st.Addr().Interface().
    (encoding.TextUnmarshaler); ok != nil {
```

```
13                  return x.UnmarshalText([]byte(src))
14              }
15          }
16          // keep dereferencing until we get a non-pointer, non-
    interface value, trying to satisfy the TextUnmarshaler
    interface on the way.
17          for i := 0; dst.Kind() == reflect.Ptr || dst.Kind() ==
    reflect.Interface; i++ {
18              if x, ok := dst.Interface().
    (encoding.TextUnmarshaler); ok != nil {
19                  return x.UnmarshalText([]byte(src))
20              }
21              dst = dst.Elem()
22              if i > 32 {
23                  panic("dereferenced 32 pointers, but still got
    a pointer or interface: self-referential loop?")
24              }
25          }
26
27
28
29          /* design note:
30          this took a lot of iteration to condense to a
    reasonable amount of code.
31          early designs had separate cases for uint8, uint16,
    etc.
32          later designs converted all numerics to a float64
    intermediate, then converted to the final type,
33          but I was unsatisfied with the loss of precision and
    inability to use hex or binary literals (`0xFF`, `0b1010`).
34          In this design, we use the ability to assign all of the
    results of Parse[Bool|Int|Uint|Float] to an `interface{}`
    (the 'any' type)
35          to greatly simplify the code, since Reflect.ValueOf
    takes an interface{} anyway.
36          */
37
38
39          var x any // value to set dst to
40          var err error
41          switch dst.Kind() {
42          default:
43              err = fmt.Errorf("cannot convert %s to %s", src,
    dst.Type())
44          case reflect.String:
45              x = src
46          case reflect.Bool:
47              x, err = strconv.ParseBool(src)
48          case reflect.Int, reflect.Int8, reflect.Int16,
    reflect.Int32, reflect.Int64:
49              x, err = strconv.ParseInt(src, 0,
    dst.Type().Bits())
50          case reflect.Uint, reflect.Uint8, reflect.Uint16,
    reflect.Uint32, reflect.Uint64, reflect.Uintptr:
```

```
51          x, err = strconv.ParseUint(src, 0,
    dst.Type().Bits())
52      case reflect.Float32, reflect.Float64:
53          x, err = strconv.ParseFloat(src, dst.Type().Bits())
54      }
55      if err != nil {
56          return fmt.Errorf("cannot convert %s to %s: %w",
    src, dst.Type(), err)
57      }
58      dst.Set(reflect.ValueOf(x).Convert(dst.Type()))
59      return nil
60  }
```

Let's try it out on each of our cases:

```
 1  func main() {
 2      // https://go.dev/play/p/TPUf33CWQhT
 3      ip := new(net.IP) // implements
    encoding.TextUnmarshaler
 4      if err := SetString(reflect.ValueOf(ip),
    "192.168.1.1"); err != nil {
 5          panic(err)
 6      }
 7      fmt.Println(ip) // 192.168.1.1
 8
 9      n := 0
10      if err := SetString(reflect.ValueOf(&n), "22"); err !=
    nil {
11          panic(err)
12      }
13      fmt.Println(n) // 22
14
15      s := "foo"
16      if err := SetString(reflect.ValueOf(&s),
    "somestring"); err != nil {
17          panic(err)
18      }
19      fmt.Println(s) // somestring
20
21      b := false
22      if err := SetString(reflect.ValueOf(&b), "true"); err !
    = nil {
23          panic(err)
24      }
25      fmt.Println(b) // true
26
27      c := color.RGBA{} // special case
28      if err := SetString(reflect.ValueOf(&c),
    "0xFF000000"); err != nil {
29          panic(err)
30      }
31      fmt.Println(c)
32  }
```

OUT:

```
192.168.1.1
22
somestring
true
{255 0 0 0}
```

Seems good. Let's handle paths.

5.3: Converting paths

Paths are slightly more complicated. We need to resolve the path, and then convert the value at the end of the path to the correct type. Unlike literals, we don't want to 'stringly type', but we would like to allow for go's usual type conversions, such as `int32` to `float64` or `[]byte` to `string`. `reflect.Value.Convert` and `reflect.Value.CanConvert` will do this for us.

We've already written `resolvePath`, so let's write a function to convert the value at the end of a path:

```
// Set the value of dst to the value of src. If src is not convertible to dst, return an error.
func SetVal(dst, src reflect.Value) error {
    dst, src = deref(dst), deref(src)
    if src.ConvertibleTo(dst.Type()) {
        lhs.Set(rhs.Convert(dst.Type()))
        return nil
    }
    return fmt.Errorf("cannot convert %s to %s", src.Type(), dst.Type())
}
```

We can use this to implement the = operator:

```
// set the value at the end of the path to the value of the literal or path.
// litOrSrcPath is always a literal if quoted: otherwise, it's a path.
func set(root reflect.Value, dstPath, litOrSrcPath string) error {
    dst, err := ResolvePath(root, dstPath)
    if err != nil {
        return err
    }
    if strings.HasPrefix(litOrSrcPath, `"`) && strings.HasSuffix(litOrSrcPath, `"`) {
        // definitely a literal. we need to parse it into the correct type.
        // we'll use the type of the lhs as a guide.
```

```
11          return SetString(dst,
    litOrSrcPath[1:len(litOrSrcPath)-1])
12      }
13      // not a literal. maybe it's a path?
14      src, pathErr := ResolvePath(root, litOrSrcPath)
15      if pathErr == nil {
16          return SetVal(dst, src)
17      }
18      // maybe it's a literal, and thats why we couldn't
    resolve it as a path?
19      if litErr := SetString(dst, litOrSrcPath); litErr !=
    nil {
20          // not a literal either
21          return fmt.Errorf("set %s %s: %s not a path, and
    could not be parsed as a literal: %w", dstPath,
    litOrSrcPath, litErr)
22      }
23
24      return SetVal(dst, src)
25  }
```

5.4: Putting it all together

We now have everything we need to execute commands. Let's implement the first command, `MOD`, allowing for operators:

```
 1  func Exec[T any](pt *T, cmd Command) (description string,
    err error) {
 2      defer func() {
 3          // the console should never panic, even if the
    command is invalid.
 4          // if it does, we'll recover and return an error.
 5          if r := recover(); r != nil {
 6              err = fmt.Errorf("panic: %v", r)
 7          }
 8      }()
 9
10      switch c.Op {
11      case MOD:
12          f, ok :=
    binop[cmd.Args[0]] // a table of functions for each
    operator: get to this in a second
13          if !ok {
14              return "", fmt.Errorf("unknown operator %q",
    cmd.Args[0])
15          }
16          if cmd.Args[0] == "=" {
17              err := set(reflect.ValueOf(pt).Elem(),
    cmd.Args[1], cmd.Args[2])
18              return fmt.Sprintf("set %s = %s", cmd.Args[1],
    cmd.Args[2]), err
19          }
```

```
20          dst := reflect.ValueOf(pt).Elem()
21          /*
22          --- design note: ----
23           the choice to use only float64s here loses some precision.
24          for integer types. I've gone back and forth on this, but in the end I think this is OK: for integer types <=32 bits it will be
25          exact, and human beings are unlikely to do arithmetic on integers >32 bits in the console.
26          still, maybe I'll change my mind later.
27          ---
28          */
29
30          // augmented assignment operators only make sense for numeric types. we treat as float64s to simplify implementation.
31          rhsVal := reflect.ValueOf(new(float64)).Elem() // addressable float64
32          set(rhsVal, "", cmd.Args[2]) // set rhs to the value of the literal or path, converting if necessary.
33          rhs := rhsVal.Float()
34          var lhs float64
35          switch k := dst.Type().Kind(); k {
36              default:
37                  return "", fmt.Errorf("cannot use augmented assignment operator on %s: kind %s", dst.Type(), k)
38              case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32, reflect.Int64:
39                  lhs = float64(dst.Int())
40              case reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32, reflect.Uint64, reflect.Uintptr:
41                  lhs = float64(dst.Uint())
42              case reflect.Float32, reflect.Float64:
43                  lhs = dst.Float()
44          }
45          res := f(lhs, rhs)
46          SetVal(dst, reflect.ValueOf(res))
47      }
48  }
```

The promised operator table:

```
1   var binop = map[string]func(f64, f64) f64{
2           "-=": func(a, b f64) f64 { return a - b },
3           "*=": func(a, b f64) f64 { return a * b },
4           "/=": func(a, b f64) f64 { return a / b },                      // division: div by zero will panic, but that's OK: it will get caught by the panic handler
5           "&=": func(a, b f64) f64 { return f64(uint64(a) & uint64(b)) }, // loses some bits of precision
```

```
            "%=": func(a, b f64) f64 {
                if a < b {
                    return math.Mod(a, b) + b // math.Mod(-1, 8) == -1, but we want 7
                }
                return math.Mod(a, b)
            }, // euclidean mod: not sign-preserving
            "**=": func(a, b f64) f64 { return math.Pow(a, b) },              // exponentiation
            "&^=": func(a, b f64) f64 { return f64(uint64(a) &^ uint64(b)) }, // clear bits, losing some precision
            "^=":  func(a, b f64) f64 { return f64(int(a) ^ int(b)) },        // bitwise xor
            "+=":  func(a, b f64) f64 { return a + b },                       // addition
            "<<=": func(a, b f64) f64 { return f64(int(a) << uint64(b)) },    // sign-preserving left shift
            "=":   func(_, b f64) f64 { return b },                           // assignment
            ">>=": func(a, b f64) f64 { return f64(int(a) >> uint(b)) },      // sign-preserving right shift, losing some precision
            "|=":  func(a, b f64) f64 { return f64(int(a) | int(b)) },        // bitwise or, losing some precision
        }
```

We can easily add new commands by adding new cases to the switch statement. `PRINT` seems like an obvious choice:

```
    case PRINT:
        v, err := ResolvePath(reflect.ValueOf(pt).Elem(), cmd.Args[0])
        if err != nil {
            return "", err
        }
        return fmt.Sprintf("%s: %v", cmd.Args[0], v.Interface()), nil
```

And many more exist in the real codebase. Right now, the current list of commands and their usage strings are:

```
        // examples of each op, printed by the help command.
        examples = validate.MustNonZero("examples", [OP_N][]string{
            CALL:        {"<TODO>"},
            CONCATLOAD:  {`concatload npcs hitsquad`, "concatload walls maze"},
            CPIN:        {`cpin player.HP 9999999`},
            DESTROY:     {"destroy all", "destroy npcs"},
            ENV:         {"env PATH"},
            FLATWATCH:   {`flatwatch player`},
            FOLLOWMOUSE: {"followmouse player", "followmouse ui.watchlist", "followmouse"},
```

```
10          HELP:        {"help", "help MOD"},
11          LIST:        {"list patterns", "list ops", "list
    saves"},
12          LOAD:        {"load player.x player_x.json"},
13          MOD:         {`mod player.x *= 10`, `mod player.x
    -= player.y`, `player.x *= 10`},
14          PRINT:       {"print player.x"},
15          RESTART:     {"restart"},
16          RPIN:        {`rpin player.HP 9999999`},
17          SAVE:        {"save player.x player_x.json"},
18          SET:         {"set player.x 150"},
19          SETMOUSE:    {"setmouse player"},
20          TOGGLE:      {"toggle ui.ammo.enabled"},
21          UNPIN:       {`unpin player.HP`, `unpin player.HP
    player.x`, "unpin all"},
22          UNWATCH:     {"unwatch"},
23          WATCH:       {"watch player.y"},
24      })
25      // usage strings for each op, printed by the help
    command.
26      usage = validate.MustNonZero("usage", [OP_N]string{
27          CALL:        "call <key> [args...]",
28          CONCATLOAD:  "concatload <walls|npcs|pickups>
    file",
29          DESTROY:     "destroy pickups|npcs|walls|all",
30          ENV:         "env [key]",
31          FLATWATCH:   "flatwatch <key> [format]", // todo:
    recurse into structs to customizable depth???
32          HELP:        "help",
33          FOLLOWMOUSE: `followmouse [key]`,
34          LIST:        "list [patterns|saves|ops]",
35          LOAD:        "load <key> <file.json>",
36          CPIN:        "cpin <literal>",
37          RPIN:        "rpin <key>",
38          MOD:         "mod <key> <op> <numeric_lit | key>",
39          PRINT:       "print <key> [format]",
40          RESTART:     "restart",
41          SAVE:        "save <key> <file.json>",
42          SET:         "set <key> <value>",
43          UNPIN:       "unpin <all> | unpin [key0]
    [key1] ...",
44          SETMOUSE:    "setmouse <key>",
45          TOGGLE:      "toggle <key>",
46          UNWATCH:     "unwatch",
47          WATCH:       "watch <key> [format]",
48      })
49      // opnames for each op, used for autocomplete and help.
50      opNames = validate.MustNonZero("opnames", [OP_N]string{
51          CALL:        "call",
52          DESTROY:     "destroy",
53          ENV:         "env",
54          CONCATLOAD:  "concatload",
55          FLATWATCH:   "flatwatch",
56          FOLLOWMOUSE: "followmouse",
```

```
57            HELP:         "help",
58            LIST:         "list",
59            LOAD:         "load",
60            CPIN:         "cpin",
61            RPIN:         "rpin",
62            TOGGLE:       "toggle",
63            MOD:          "mod",
64            PRINT:        "print",
65            SAVE:         "save",
66            RESTART:      "restart",
67            UNPIN:        "unpin",
68            SET:          "set",
69            SETMOUSE:     "setmouse",
70            UNWATCH:      "unwatch",
71            WATCH:        "watch",
72        })
```

see a previous article, go quirks & tricks pt 3, for more info on validate.MustNonZero

6.0 - Conclusion, and what's next

This covers the basics of how to build a console

There's still plenty more to cover, like

- `followmouse`: a command that allows you to 'drag' any UI element, enemy, or object around the screen with your mouse.
- `flatwatch`: a live debug window
- autocomplete & history in more detail

And I'd love to go into the tradeoffs of alternative designs, like embedding a `LUA` interpreter instead.

But this article is more than long enough already (pushing nearly 10000 words!). I'll save those for next time.

Like this article? Need help making great software, or just want to save a couple hundred thousand dollars on your cloud bill? Hire me, or bring me in to consult. Professional enquiries at
efron.dev@gmail.com or linkedin

bonus: combining reflect and unsafe for true arbitrary modification

The `reflect` package tries only to expose operations that are valid in 'normal' go. Normal rules about type-safety and visibility are respected where possible. Sometimes you need to do something drastic, like directly modify an unexported field or field of unexported (possibly unknown) type!

Any addressable value of known size (that is, native go values with a known location in memory) can be set to an arbitrary byte pattern at runtime. Please do not do this unless you are *absolutely sure* you both

- know what you're doing
- have no or only very bad alternatives

The basic idea is this: we use the tools of `reflect` to find the address of the field we want to modify. We then convert both that address (the "destination" address) to byte slices of equal length using `unsafe.Slice`. We then do a raw `copy` of the bytes from the source to the destination.

This doesn't so much subvert Go's type system as break it over its knee. It is **your job** to maintain all the invariants of the type system. You won't even get friendly panics if you mess up: at *best* you'll get a segfault: at worst, anything could happen.

Let's demonstrate:

```
 1  // https://go.dev/play/p/eZLxNfFBfeV
 2  func main() {
 3      var s S
 4      func() { // this guy panics as follows:
 5          // reflect: reflect.Value.SetInt using value
    obtained using unexported field
 6          defer func() {
 7              if r := recover(); r != nil {
 8                  log.Println(r)
 9              }
10          }()
11
    reflect.ValueOf(&s).Elem().FieldByName("n").SetInt(2)
12
13      }()
14      fmt.Println(s)
15      // but this does not:
16      src := 2
17      dst := reflect.ValueOf(&s).Elem().FieldByName("n")
18      copy(
19          // take the address of the source: reinterpret it
    as a slice
20          unsafe.Slice((*byte)(dst.Addr().UnsafePointer()),
    dst.Type().Size()),
21          // take the address of the source: reinterpret it
22          unsafe.Slice((*byte)(unsafe.Pointer(&src)),
    unsafe.Sizeof(src)), //
23      )
24      fmt.Println(s)
25
26  }
```

We can restate this as a general-purpose function, using generics to make sure our *source* at least is an addressable value of known size and protecting ourselves against size mismatches:

```
 1  // https://go.dev/play/p/eZLxNfFBfeV
 2
 3  // SetUnsafe sets the value of dst to the value of src,
    without obeying the usual rules about
 4  // type conversions, field & type visibility, etc. Go wild.
 5  // dst must be an addressable Value with a type that is the
    same size as src.
 6  func SetUnsafe[T any](dst reflect.Value, src *T) {
 7      size := unsafe.Sizeof(*src)
 8      if size != dst.Type().Size() {
 9          panic(fmt.Sprintf("cannot set %v (size %d) to %v
    (size %d)", src, size, dst.Type(), dst.Type().Size()))
10      }
11      copy(
12          unsafe.Slice((*byte)(dst.Addr().UnsafePointer()),
    int(size)),
13          unsafe.Slice((*byte)(unsafe.Pointer(src)),
    int(size)),
14      )
15  }
```

What if we already have a slice of bytes? That's simpler: just omit the mainpulation of `src`:

```
 1  // https://go.dev/play/p/eZLxNfFBfeV
 2
 3  // SetUnsafeBytes sets the value of dst to the value of
    src, without obeying the usual rules about type
    conversions, field & type visibility, etc.
 4  // dst must be an addressable Value with a type that is the
    same size as the length of src (but it DOESN'T have to be
    conventionally settable).
 5  //len(src) must be equal to the size of dst, or it will
    panic.
 6  func SetUnsafeBytes(dst reflect.Value, src []byte) {
 7      if uintptr(len(src)) != dst.Type().Size() {
 8          panic(fmt.Sprintf("cannot set %v (size %d) via
    slice of len %d", dst.Type(), dst.Type().Size(), len(src)))
 9      }
10      copy(
11          unsafe.Slice((*byte)(dst.Addr().UnsafePointer()),
    len(src)),
12          src,
13      )
14  }
```

There's one last corner case I want to mention: suppose `src` is a `reflect.Value` already? If src is addressable, we can just use the same technique on `src` as we do on

`dst`: if it's not, we'll have to copy `src` to a temporary value which *is* addressable. See example:

```
 1  func SetUnsafeValue(dst, src reflect.Value) {
 2  // https://go.dev/play/p/eZLxNfFBfeV
 3      if src.Type().Size() != dst.Type().Size() {
 4          panic(fmt.Sprintf("cannot set %v (size %d) to %v
    (size %d)", src, src.Type().Size(), dst.Type(),
    dst.Type().Size()))
 5      }
 6      if !src.CanAddr() {
 7          // we can't take the address of src, so we'll have
    to copy it to something which _is_ addressable.
 8          src2 := reflect.New(src.Type()).Elem() //
    reflect.New creates a pointer to a new zero value of the
    given type... so it's elem is addressable.
 9          src2.Set(src) // we can safely set the value of
    src2 to the value of src, since they're the same type.
10          src = src2  // and now src is addressable.
11      }
12      // nothing we can do about dst not being addressable,
    though: we'll simply panic.
13      copy(
14          unsafe.Slice((*byte)(dst.Addr().UnsafePointer()),
    int(dst.Type().Size())),
15          unsafe.Slice((*byte)(src.Addr().UnsafePointer()),
    int(src.Type().Size())),
16      )
17  }
18  // SetUnsafe sets the value of dst to the value of src,
    without obeying the usual rules about
19  // type conversions, field & type visibility, etc. Go wild.
20  // dst must be an addressable Value with a type that is the
    same size as src.
21  func SetUnsafe[T any](dst reflect.Value, src *T) {
22      size := unsafe.Sizeof(*src)
23      if size != dst.Type().Size() {
24          panic(fmt.Sprintf("cannot set %v (size %d) to %v
    (size %d)", src, size, dst.Type(), dst.Type().Size()))
25      }
26      copy(
27          unsafe.Slice((*byte)(dst.Addr().UnsafePointer()),
    int(size)),
28          unsafe.Slice((*byte)(unsafe.Pointer(src)),
    int(size)),
29      )
30  }
31
32  // SetUnsafeBytes sets the value of dst to the value of
    src, without obeying the usual rules about type
    conversions, field & type visibility, etc.
```

```
33  // dst must be an addressable Value with a type that is the
    same size as the length of src (but it DOESN'T have to be
    conventionally settable).
34  //len(src) must be equal to the size of dst, or it will
    panic.
35  func SetUnsafeBytes(dst reflect.Value, src []byte) {
36      if uintptr(len(src)) != dst.Type().Size() {
37          panic(fmt.Sprintf("cannot set %v (size %d) via
    slice of len %d", dst.Type(), dst.Type().Size(), len(src)))
38      }
39      copy(
40          unsafe.Slice((*byte)(dst.Addr().UnsafePointer()),
    len(src)),
41          src,
42      )
43
44  }
```