# zygomys: embedded scripting toolkit for Go

16 March 2016

Jason E. Aten, Ph.D.
Principal Engineer, Betable.com

## Snoopy versus the Red Baron

- imagine Snoopy comes back with friends...

## data model for a plane formation, in Go

```go
// Go:
type Plane struct {
    Number   int
    Friends []Flyer
}
type Weather struct {
    Type    string // "sunny", "stormy", ...
}
type Flyer interface {
    Fly(w *Weather) (string, error) // flight is informed by the current Weather
}
type Snoopy struct {
    Plane
    Cry   string
}
func (p *Snoopy) Fly(w *Weather) (string, error) {
    w.Type = "VERY " + w.Type // side-effect, for demo purposes
    s := fmt.Sprintf("Snoopy sees weather '%s', cries '%s'", w.Type, p.Cry)
    fmt.Println(s)
    for _, flyer := range p.Friends { flyer.Fly(w) }
    return s, nil
}
```

- somewhat contrived. slice of interface, embedded struct handling

## Imagine Snoopy has Friends who Fly

```
// Go:
// https://en.wikipedia.org/wiki/McDonnell_Douglas_F/A-18_Hornet (1978 - present)
type Hornet struct {
    Plane
    Mass     float64
}
// https://en.wikipedia.org/wiki/Grumman_F6F_Hellcat (1943 - 1960)
type Hellcat struct {
    Plane
}
func (b *Hornet) Fly(w *Weather) (s string, err error) {
    fmt.Printf("Hornet.Fly() called. I see weather %v\n", w.Type)
    return
}
func (b *Hellcat) Fly(w *Weather) (s string, err error) {
    fmt.Printf("Hellcat.Fly() called. I see weather %v\n", w.Type)
    return
}
// give Weather a method to list
func(w *Weather) IsSunny() bool {
       return w.Type == "sunny"
}
```

with the data model in mind, lets interact with it using zygo...

## Make it rain

```
zygo> (def w (weather type:"stormy"))

zygo> w
 (weather type:"stormy")

zygo> (fieldls w)
["Type string"]

zygo> (methodls w)
["IsSunny func(*zygo.Weather) bool"]

zygo>
zygo> (type? w)   //  run-time type query
"weather"

zygo> (str weather)  // stringification, shows the Go (shadow struct) type with package prefix
"zygo.Weather"

zygo>
```

## Bring in some planes, in formation

```
zygo> (def he (hellcat speed:567))
zygo> (def ho (hornet SpanCm:12))
zygo> (def snoopdog (snoopy friends:[he ho] cry:"Curse you, Red Baron!" number:320))
zygo>
```

- in three lines we've instantiated and configured 3 Go structs in a tree
- with one more line, call a Go method on that Go struct, from zygo. When Snoopy Fly()s, so do his friends.

```
zygo>
zygo> (_method snoopdog Fly: w)
Snoopy sees weather 'VERY stormy', cries 'Curse you, Red Baron!'
Hellcat.Fly() called. I see weather VERY stormy
Hornet.Fly() called. I see weather VERY stormy
["Snoopy sees weather 'VERY stormy', cries 'Curse you, Red Baron!'" nil]
zygo>
```

- (_method) is special -- hence the underscore -- as it crosses system boundaries, calling into compiled Go code.
- can be sandboxed

## we've just been interacting/scripting Go data and methods...

## yeah, reflection is pretty cool

## zygomys - a scripting toolkit for Go

## zygomys -- what's in a name?

- zygo means union (yoke in Greek; the zygote was the first cell that was you).
- mys means mouse
- this is a little mouse of a language
- bonus: a "pocket gopher" known as Zygogeomys trichopus. Our mascot, "Ziggy".
- this is the union of lisp and Go. In a small cute package.
- let's use the shorter `"zygo"` for the language, when speaking aloud.

"The Michoacan pocket gopher is a small animal with short, dense, black, lustrous fur... It is docile when caught, making no attempt to bite as do other pocket gophers."
-- en.wikipedia.org/wiki/Michoacan_pocket_gopher (https://en.wikipedia.org/wiki/Michoacan_pocket_gopher)

## Getting started: How to embed the REPL in your code

See github.com/glycerine/zygomys/blob/master/cmd/zygo/main.go (https://github.com/glycerine/zygomys/blob/master/cmd/zygo/main.go). Just three steps.

```
import (
...
    zygo "github.com/glycerine/zygomys/repl"
)
func main() {
    // (1) configure it
    // See library configuration convention: https://github.com/glycerine/configs-in-golang
    cfg := zygo.NewGlispConfig("zygo")

    // (2) register your Go structs; give them a nickname
    // here we register snoopy as a handle to Go struct &Snoopy{}
    zygo.GoStructRegistry.RegisterUserdef("snoopy",
        &zygo.RegisteredType{GenDefMap: true,
            Factory: func(env *zygo.Glisp) (interface{}, error) {
                return &Snoopy{}, nil
            }}, true)

    // (3) run the zygo repl
    // -- the library does all the heavy lifting.
    zygo.ReplMain(cfg)
}
```

## architecture / overview of design

- a) lexer produces tokens
- b) parser produces lists and arrays of symbols
- c) macros run at definition type
- d) codegen produces s-expression byte-code
- e) a simple virtual machine executes s-expression byte-code. User's functions run.

## Let's see more zygo code

- hashmaps can define arbitrary records; with or without attached Go shadow structs

```
zygo> // define a hashmap with four key-value pairs:
zygo> (def  hsh  (hash a:44 b:55 c:77 d:99))
 (hash a:44 b:55 c:77 d:99)
zygo>
zygo> // define a string to concat stuff to
zygo> (def s "")
""
```

we range through the hashmap, hsh, like this:

```
zygo> // iterate over the hashmap hsh, adding to s
zygo> (range  k  v  hsh
...        /*body of loop starts -- here the body is this set call*/
...        (set s (concat s " " (str k) "-maps->" (str v) "\n")))
zygo>
zygo> s
" a-maps->44\n b-maps->55\n c-maps->77\n d-maps->99\n"
zygo>
```

## records

- records are hash tables with a name. All hash tables preserve key-order.

```
/* Harry Potter goes West in 'Hogwild!'. We'll build and then query this structure:
  (ranch
       cowboy:"Harry"
       cowgirl:"Hermonie"
       bunk1: (bunkhouse
           bed1:"Lucius"
           bed2:"Dumbledore"
           closet1: (closet
               broom:"Nimbus2k" ) ) ) )
*/
zygo> (defmap ranch)
zygo> (def hogwild (ranch cowboy:"Harry" cowgirl:"Hermonie"))
zygo> (defmap bunkhouse)
zygo> (hset hogwild bunk1:(bunkhouse bed1:"Lucius" bed2: "Dumbledore"))
zygo> (defmap closet)
zygo> (hset (:bunk1 hogwild) closet1:(closet broom:"Nimbus2k"))

zygo> (hget (hget (hget hogwild bunk1:) closet1:) broom:)  // step by step query
"Nimbus2k"
zygo> (-> hogwild bunk1: closet1: broom:)  // clojure style threading
"Nimbus2k"
```

## arrays (slices) - can hold any Sexp

```
zygo> (def ar [4 5 6 7])
[4 5 6 7]
zygo> (aget ar 0)
4
zygo> (aget ar 1)
5
zygo> (aset ar 1 999)
zygo> ar
[4 999 6 7]
zygo>
```

## aims

- interactive, but also aim to eventually compile-down to Go
- blending Go and lisp
- I built it for myself
- technically interesting about the zygo implementation:
- using goroutines as coroutines to get pause-able parsing. avoids the O(n^2) trap when reading multi-line user-typed input. Call for more input from many points; inversion of select loop -- exit when you've got another line of input tokens. See repl/parser.go.
- if you haven't discovered how to do conditional sends on a channel yet, examples inside `github.com/glycerine/zygomys/repl/parser.go`.

## status: hard parts that are already done

- script calls to existing Go functions using existing Go structs. Uses reflection.
- Go-style for-loops. Nest-able. With break to label, and continue to label.
- eval
- sandbox / restrict filesystem access
- full closures with lexical scope
- adjust lisp syntax to be Go compatible: % for quoting, 'a' for characters.
- higher order functions.

## classic lisp style - list processing

```
// foldr: right fold is a classic higher order function
//       It runs a function over each element in a list.
//
//  lst: pair list, the input
//  fun: processes one element in the list
//  acc: the accumulated result, the output
//
(defn foldr [lst fun acc]
    (cond                  // cond is zygo's if-then-else.
        (empty? lst) acc   // return acculated output if no more input.
            (fun           // else call fun on the head of lst
               (car lst)
               (foldr (cdr lst) fun acc)) // recursive call on the tail of the input
    )
)
```

- see the closure tests in github.com/glycerine/zygomys/blob/master/tests/closure.zy

  (https://github.com/glycerine/zygomys/blob/master/tests/closure.zy)

## there is also an infix interface

- anything inside curly braces is infix parsed. Can mix in function calls. Math becomes more readable. Uses a Pratt parser.

```
zygo> { a = 10 }
10
zygo> { b = 12 }
12
zygo> { a + b * 2 }
34
zygo> { a + b * 2 / .5}
58
zygo> (defn cube [x] (* x x x)) // or (defn cube [x] {x*x*x})
zygo> a
10
zygo> b
12
zygo> cube
(defn cube [x] (* x x x))
zygo> (cube b)
1728
zygo> { a + (cube b) ** 2}  // L ** R means raise L to the power R.
2985994
zygo>
```

## the basic Go API: adding compiled functions to zygo

```
// Go: see https://github.com/glycerine/zygomys/blob/master/repl/functions.go
//
func FirstFunction(env *Glisp, name string, args []Sexp) (Sexp, error) {
    if len(args) != 1 {
        return SexpNull, WrongNargs
    }
    switch expr := args[0].(type) {
    case *SexpPair:
        return expr.Head, nil
    case *SexpArray:
        if len(expr.Val) > 0 {
            return expr.Val[0], nil
        }
        return SexpNull, fmt.Errorf("first called on empty array")
    }
    return SexpNull, WrongType
}
```

```
zygo> (first [5 6 7])
5
zygo> (first (list "hi" "there"))
"hi"
zygo>
```

## json / msgpack support

See the top of `github.com/glycerine/zygomys/repl/jsonmsgp.go` for a guide.

```
 Conversion map:

 Go map[string]interface{}  <--(1)--> lisp
   ^                                  ^ |
   |                                 /  |
  (2)   ------------ (4) -----------/  (5)
   |   /                                |
   V  V                                 V
 msgpack <--(3)--> go struct, strongly typed

(1) we provide these herein; see jsonmsgp_test.go too.
     (a) SexpToGo()
     (b) GoToSexp()
(2) provided by ugorji/go/codec; see examples also herein
     (a) MsgpackToGo() / JsonToGo()
     (b) GoToMsgpack() / GoToJson()
(3) provided by tinylib/msgp, and by ugorji/go/codec
     by using pre-compiled or just decoding into an instance
     of the struct.
(4) see herein
     (a) SexpToMsgpack() and SexpToJson()
     (b) MsgpackToSexp(); uses (4) = (2) + (1)
(5) The SexpToGoStructs() and ToGoFunction() in this file.
```

## the fundamental Sexp types

```
type Sexp interface {
    SexpString(indent int) string
    Type() *RegisteredType
}
```

- SexpNull (actually a value; an instance of the SexpSentinel type)
- SexpSymbol (variable and function names; symbol table entries)
- SexpPair (linked lists)
- SexpArray (slices)
- SexpHash (hash table; keys and values are any Sexp, key ordering preserved)

## type system: work in progress...

- always manifestly typed: a variable points to a value that knows its own type.
- add-on: optional static type system -- enforced at definition time -- is half implemented
- to follow status, `github.com/glycerine/zygomys/tests/decl_fun.zy`
- struct declarations done, function type declarations with (func) not yet done.
- the static typing aims for compatibility with Go types (to enable compile-down)

```
// struct type definition
// Note the `e` numbers for evolution (like protobufs); + the deprecated notation
(struct Car [
 (field              Id: int64        e:0       gotags:`json:"id",name:"id"` )
 (field            Name: string       e:1       gotags:`json:"name"`         )
 (field        BadApple: string       e:2       deprecated:true                  )
 (field     SliceOfLife: ([]string)   e:3 )     // slice of string, matching Go's syntax.
 (field  PointerSisters: (* int64)    e:4 )     // pointer to an int64
 (field        OtherCar: (* Car)      e:5 )     // pointer to a Car struct
 ])
```

## zygo uses

- scripting, pause and interact with data
- configuration language
- Eventually... multi-core friendly scripting. (Channels now, but no select yet).
- Eventually... leverage Go's multicore strength for exploratory configuration, data analysis and scripting. (I love R for productivity, Go for production).

## it's the future ... dream big

- constraint/logic programming syntax (use sigils)
- model checking syntax and checker (Ordered-Binary-Decision-Diagram based)
- port TLA+ model checker to Go, with zygomys interface
- unification engine for type system experiments

## final thoughts

- interpreters taught me the incredible power of a test-driven approach
- tests are simply language fragments
- fork it and experiment!

## credits

The ancestor dialect of zygomys, github.com/zhemao/glisp (https://github.com/zhemao/glisp) Glisp, was designed and implemented by Howard Mao zhehaomao.com/ (https://zhehaomao.com/).

Thanks Howard!

## Thank you

Jason E. Aten, Ph.D.
Principal Engineer, Betable.com
j.e.aten@gmail.com (mailto:j.e.aten@gmail.com)

@jasonaten_ (http://twitter.com/jasonaten_)

https://github.com/glycerine/zygomys (https://github.com/glycerine/zygomys)

(#ZgotmplZ) github.com/glycerine/zygomys/wiki. (https://github.com/glycerine/zygomys/wiki.) The wiki has details, examples, and discussion.