Prototype your design!

Robert Griesemer

dotGo 2016, Paris

Getting to good software design

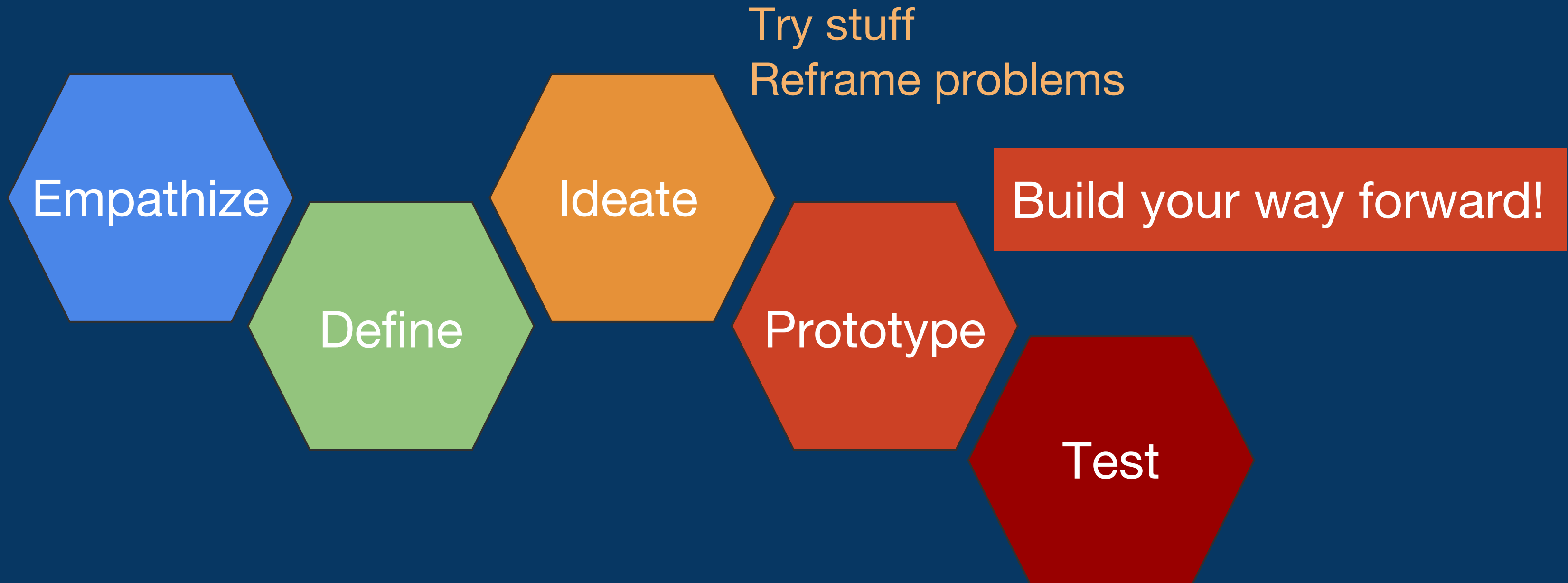
- Literature is full of design paradigms

- Usually involves
  - Design docs
  - Feedback from reviewers
  - Iterative process
  - etc.

- Often a “dry” exercise
  - No software is created until design is “completed”

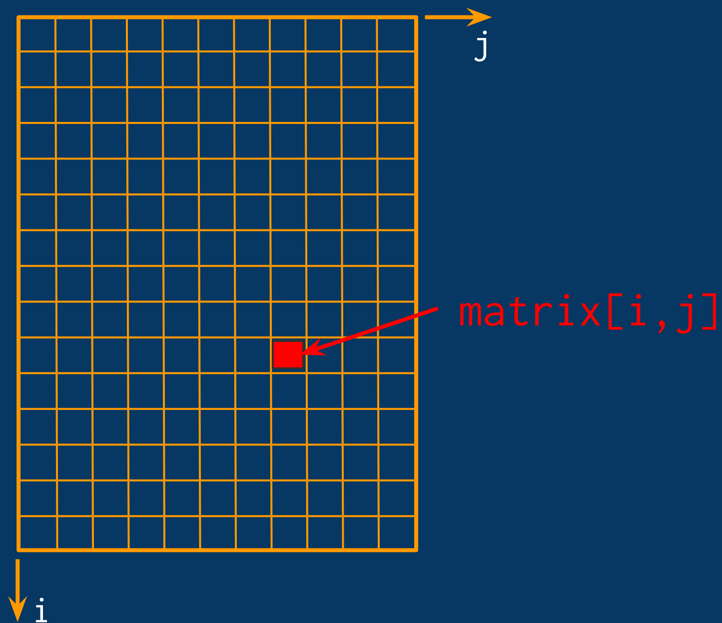
How can we tell if we have a good design?

Elsewhere, design thinking requires prototyping



From Design School Stanford: http://dschool.stanford.edu/

Example: Designing Go support for numerical apps



Multi-dimensional slices for Go (issue #6282)

```
var matrix [,]float64

matrix = make([,]float64, 15, 11)
```

High-level goals

1. More readable code
2. Great performance

Many open questions:

- Which primitive operations?
- What implementation?
- What notation?
- etc.

Observation

We can implement many aspects of multi-dimensional slices in Go now:

- Slice representation
  - ⇨ Define an (abstract data) type
- Creation, access, mutation
  - ⇨ Define appropriate methods on that type

A Go implementation allows us to explore our design.

Key missing feature: Nice notation

The work-around, accessor methods for multi-dim. index expressions

```
m.At(i, j)
m.AtSet(i, j, x)
```

makes numerical code clunky, perhaps even unreadable:

```
c.AtSet(i, j, a.At(i, k) * b.At(k, ind.At(j)))
```

instead of

```
c[i, j] = a[i, k] * b[k, ind[j]]
```

How can we get around the notation problem?

- Declare it not a problem
  - Not an option

- Change the Go language for the experiment
  - Too costly

- Rewrite the source code:

```
a[i, j]  ⇨  a.At(i, j)
```

We can do this by hand, or automatically, via a source-to-source rewriter.

A prototype allows us to explore the design space.

Design the prototype

- Allow index operators as method names
  - `[]` indexed getter
  - `[]=` indexed setter (assignment)
  - `+` addition (for illustration purposes only)

- Permit multiple indices in index expressions

- Semantics
  - `x[i]` means `x.[](i)`
  - `x[i, j]` means `x.[](i, j)`
  - `x[i, j, k] = y` means `x.[]=(i, j, k, y)`
  - `x + y` means `x.+(y)`

Implement the prototype

- Rename method names into valid Go identifiers
  - `[]` ➪ `AT__`
  - `[]=` ➪ `ATSET__`
  - `+` ➪ `ADD__`
- Rewrite index expressions into valid Go method calls
  - `x[i, j]` ➪ `x.AT__(i, j)`
  - `x[i, j] = y` ➪ `x.ATSET__(i, j, y)`
  - `x + y` ➪ `x.ADD__(y)`
- To rewrite source, rewrite syntax tree
  - original source → go/parser → rewriter → go/printer → rewritten source

Example: Rewrite of + method

BEFORE

```
type Point struct { X, Y int }

func (a Point) +(b Point) Point {
    return Point{…}
}

var a, b, c Point
c := a + b
```

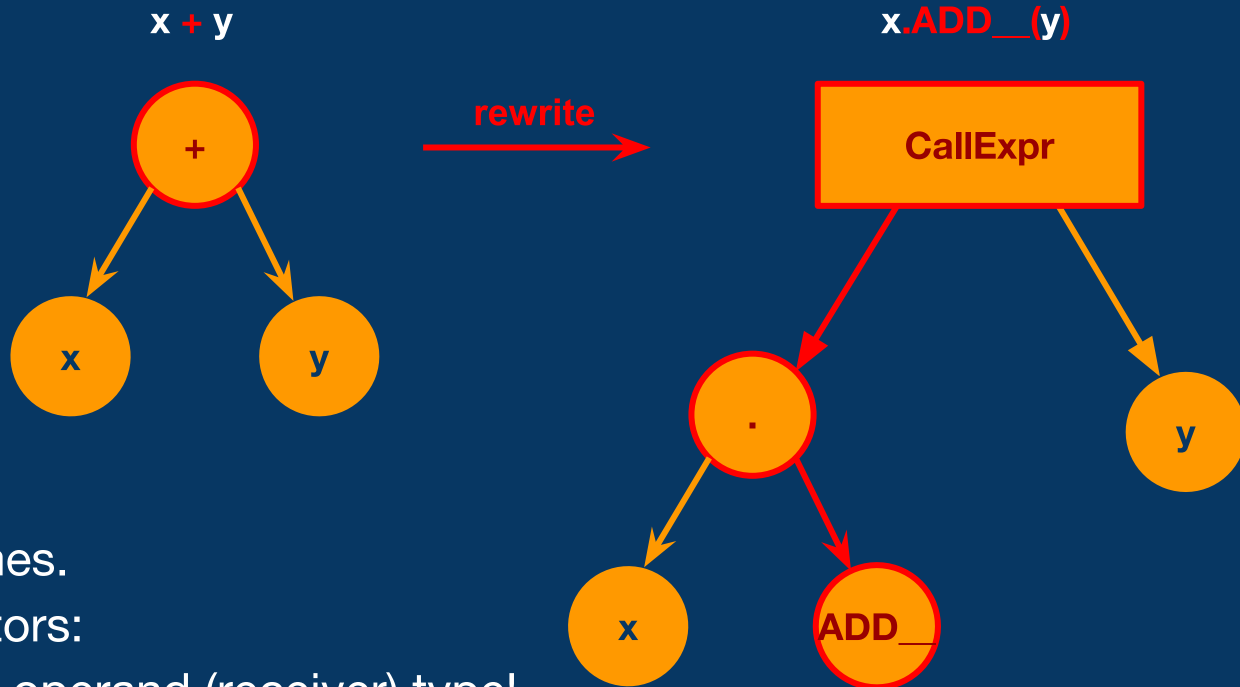
AFTER

```
type Point struct { X, Y int }

func (a Point) ADD__(b Point) Point {
    return Point{…}
}

var a, b, c Point
c := a.ADD__(b)
```

Syntax tree rewriting



Trivial for method names.

Not so easy for operators:

Need to know left operand (receiver) type!

Type-checking to the rescue

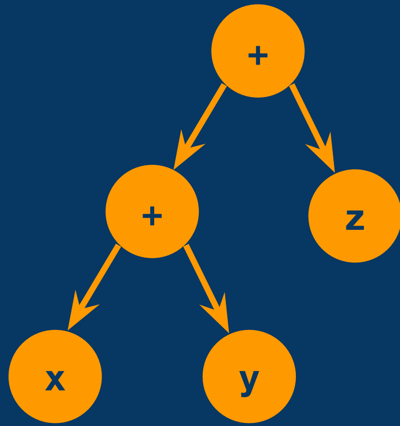
Approach:

1. Use go/types to determine operands types
2. Rewrite `x + y` if type of `x` has `ADD__` method

This works also for indexing operators.

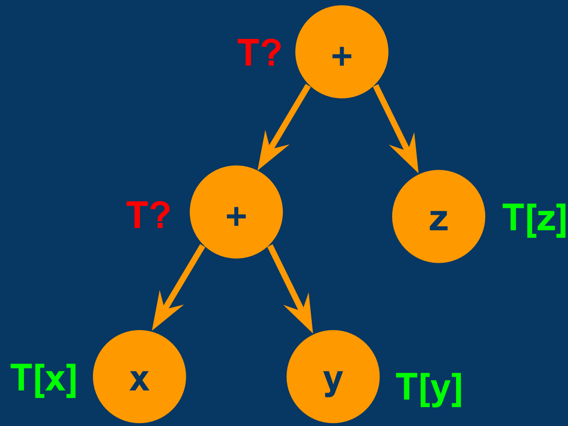
Syntax tree for x + y + z after parsing

We have no type information.



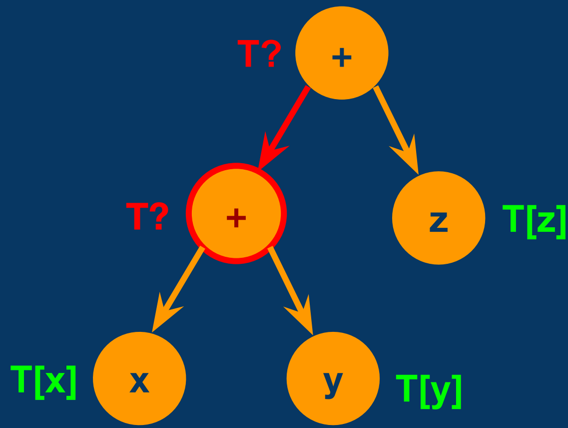
After type-checking

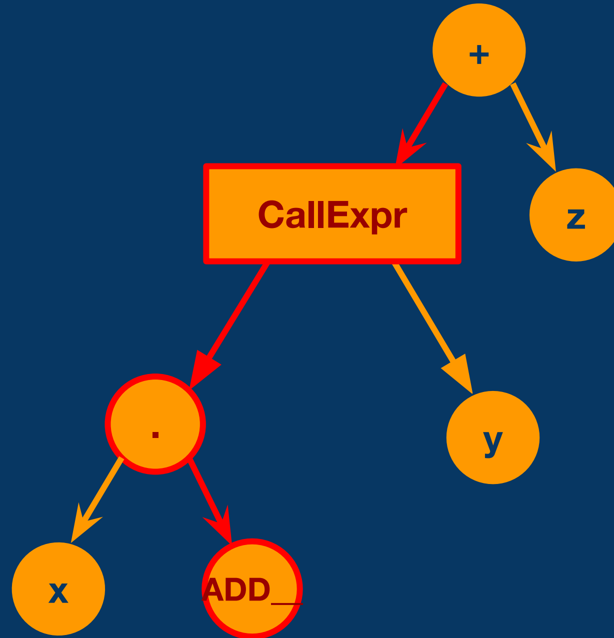
Several unknown types;
assume it’s because `x + y` should be `x.ADD__(y)`.



Rewrite where we can

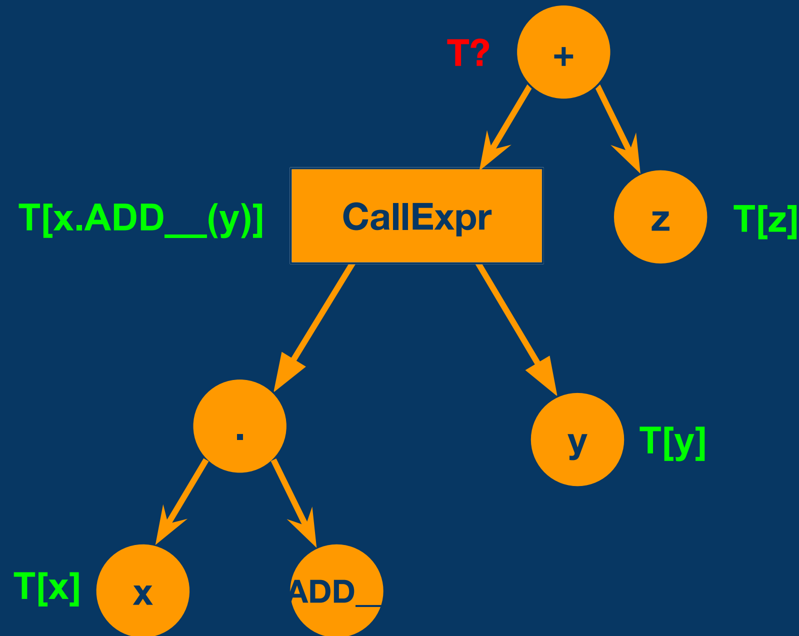
Replace `x + y` with `x.ADD__(y)`
if type of `x` implements `ADD__`.





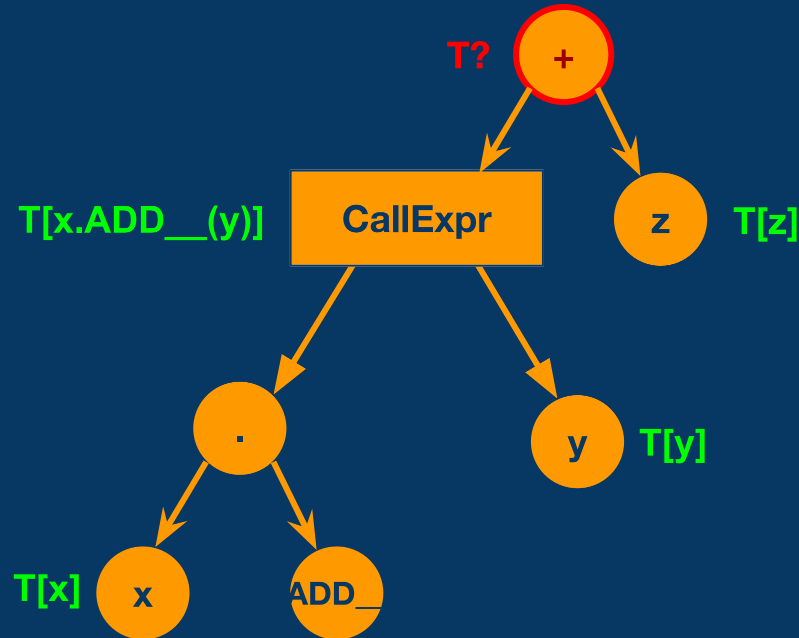
… and type-check again

Still have unknown type;
do another round.

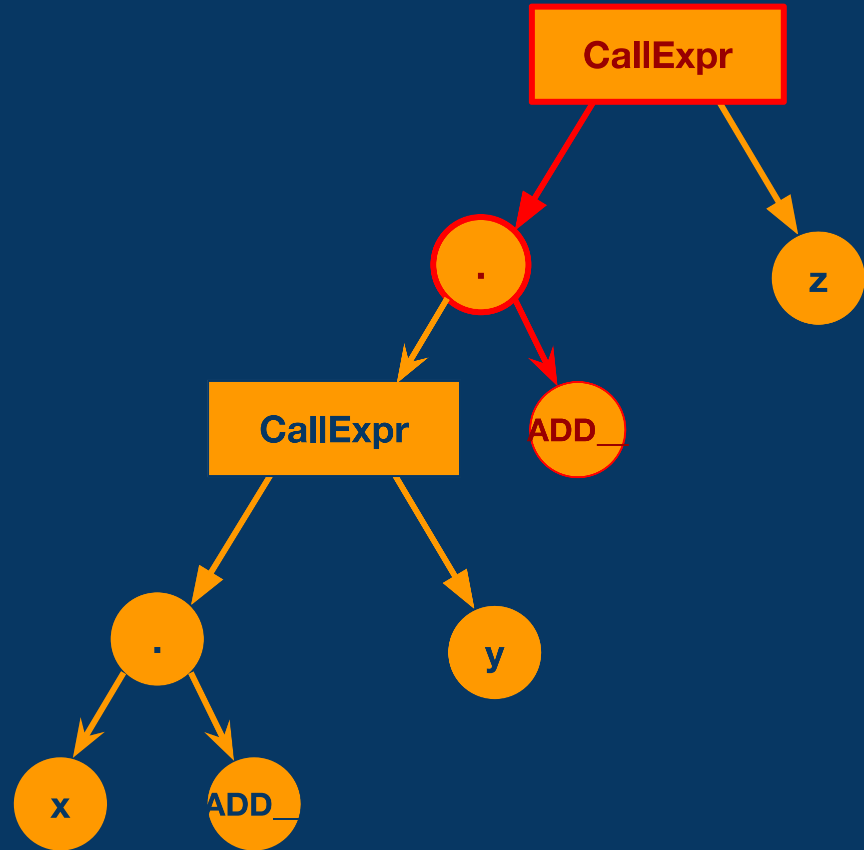


One more time: Determine what to rewrite

Replace `(x.ADD__(y)) + z` with
`(x.ADD__(y)).ADD__(z)`

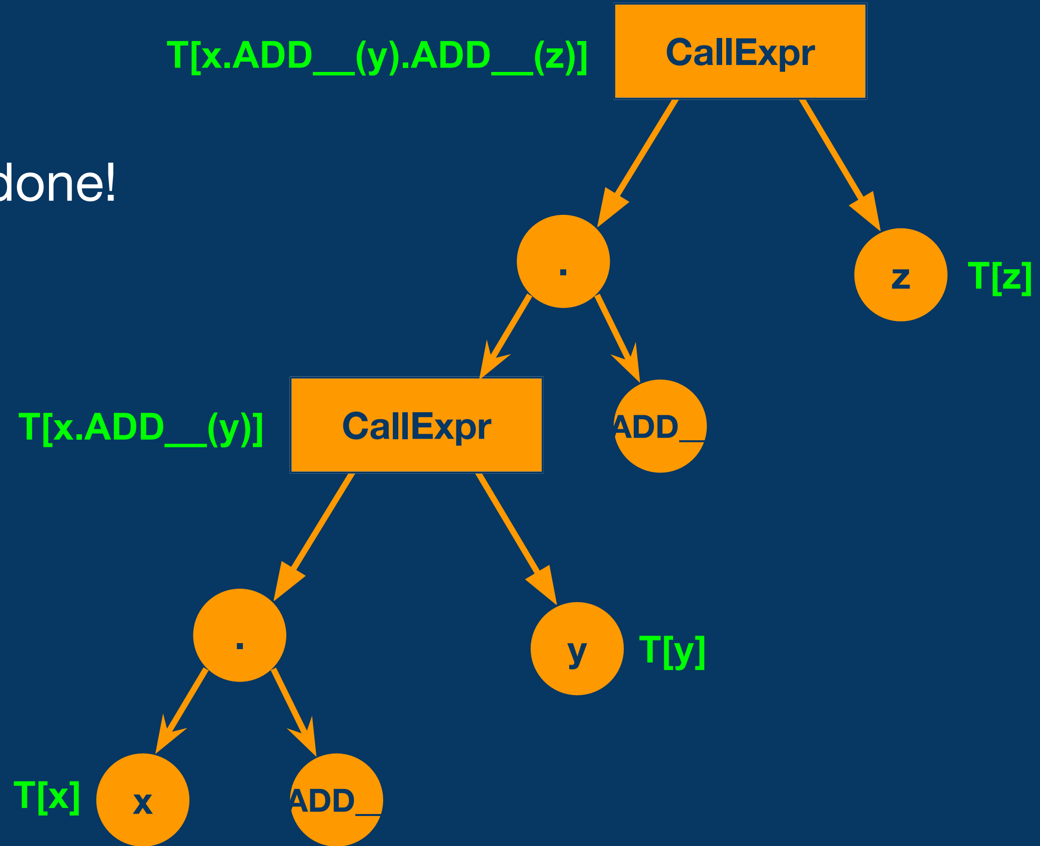


… rewrite



… and type-check

All types are known; we are done!



A concrete implementation allows us to judge a design.

An implementation of a 2D “slice”

```
type Matrix struct {
    array       []float64
    len, stride [2]int
}

func NewMatrix(n, m int) *Matrix
func (m *Matrix) []  (i, j int) float64
func (m *Matrix) []= (i, j int, x float64)
…
```

Easily generalized to other (1, 2, 3, …) dimensions.

Core of (textbook) Matrix multiplication

BEFORE

```
for i := 0; i < n; i++ {
    for j := 0; j < p; j++ {
        var t float64
        for k := 0; k < m; k++ {
            t += a[i, k] *
                 b[k, j]
        }
        c[i, j] = t
    }
}
```

AFTER

```
for i := 0; i < n; i++ {
    for j := 0; j < p; j++ {
        var t float64
        for k := 0; k < m; k++ {
            t += a.AT__(i, k) *
                 b.AT__(k, j)
        }
        c.ATSET__(i, j, t)
    }
}
```

Prototyping raises design questions we didn’t even know we should be asking.

Are index operator methods good enough?

If the prototype works well, do we even need more?

Plenty of stuff to think about …

Conclusion

- Go is a fantastic language for prototyping.
- Prototyping allows us to build our way to good design.
- If we can prototype language changes, we can prototype anything.

Plan to throw one away;
you will, anyhow.

F.P. Brooks, The Mythical Man-Month, 1975.

Thank you!

https://github.com/griesemer/dotGo2016/