Gomacro: code generation made easy and fun

Massimiliano Ghilardi

Golab, Florence

2018-10-22

Summary

- What is code generation
- Why it’s useful
- Manipulating Go source code (AST)
- Pros & cons
- Better solutions?
- Examples with 3rd party libraries
- Examples with gomacro
- Live demo
- Questions

What is code generation?

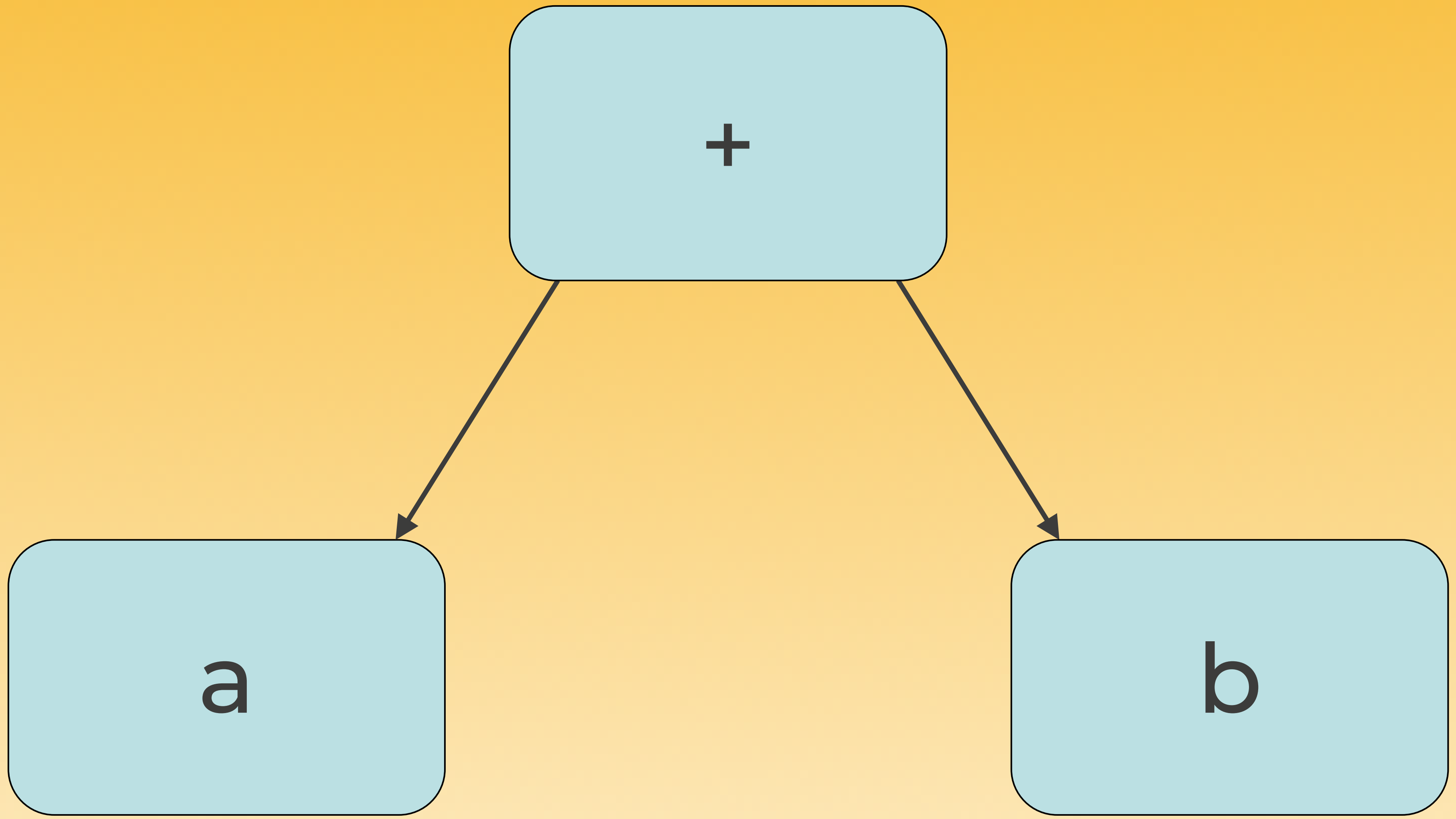
- Programs that output code

Useful for:

- Generate long, repetitive code
- Generate bindings and marshal / unmarshal functions
- Convert higher-level, compact code into lower-level, efficient code

Manipulating Go source code

Abstract Syntax Tree (AST): a+b



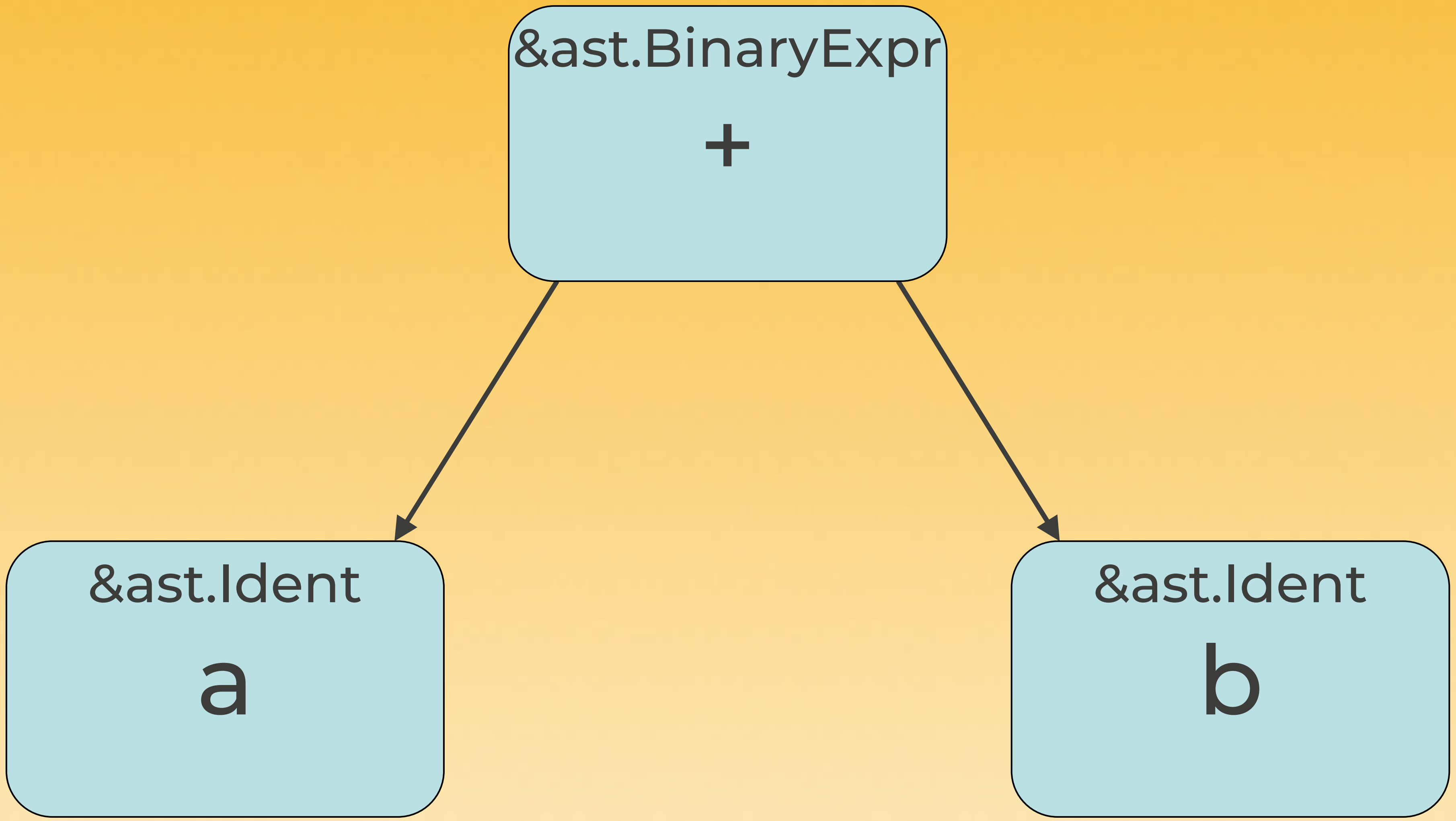
+

a

b

Manipulating Go source code

Abstract Syntax Tree (AST): a+b



&ast.BinaryExpr

+

&ast.Ident

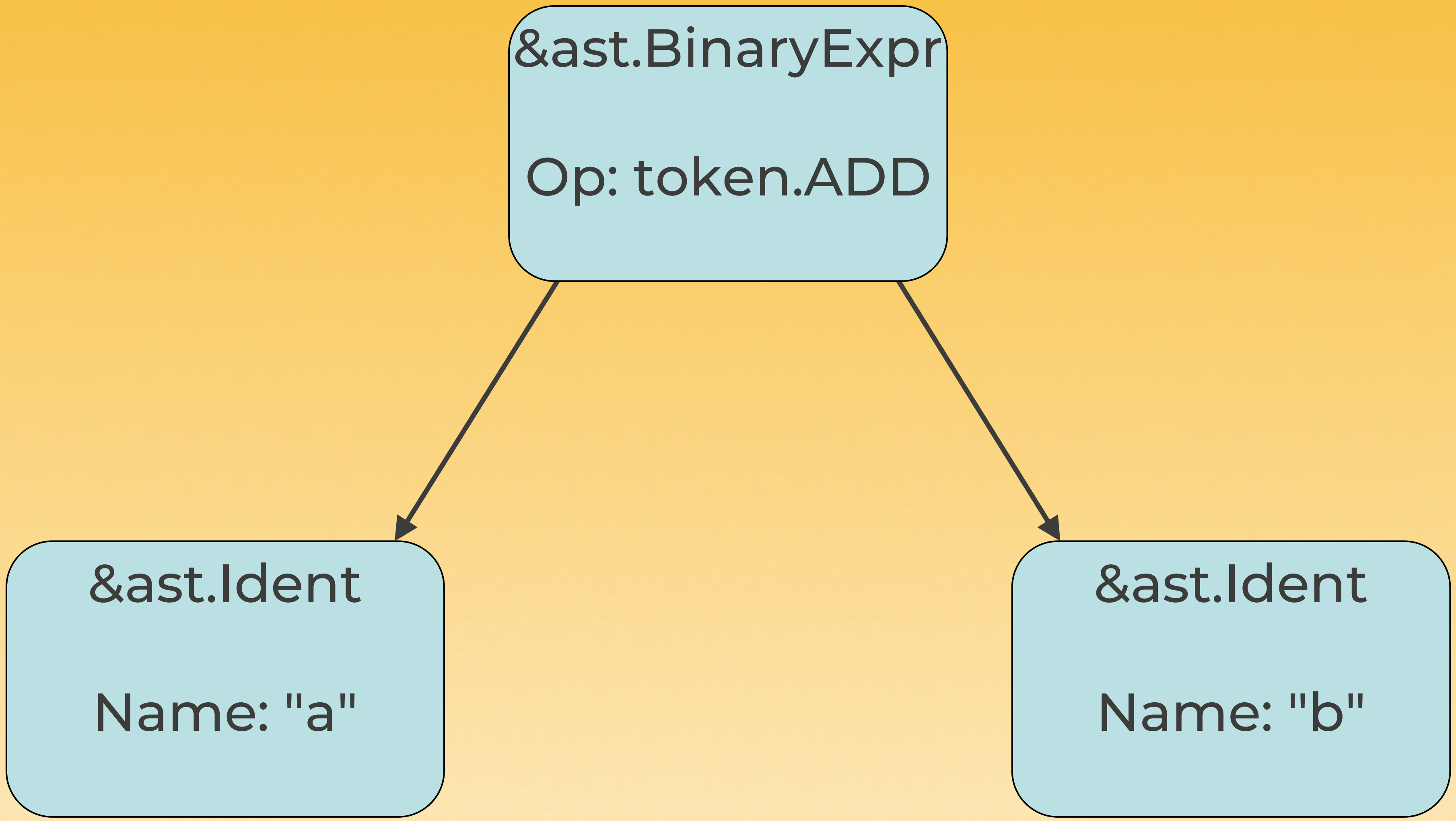
a

&ast.Ident

b

Manipulating Go source code

Abstract Syntax Tree (AST): a+b



&ast.BinaryExpr

Op: token.ADD

&ast.Ident

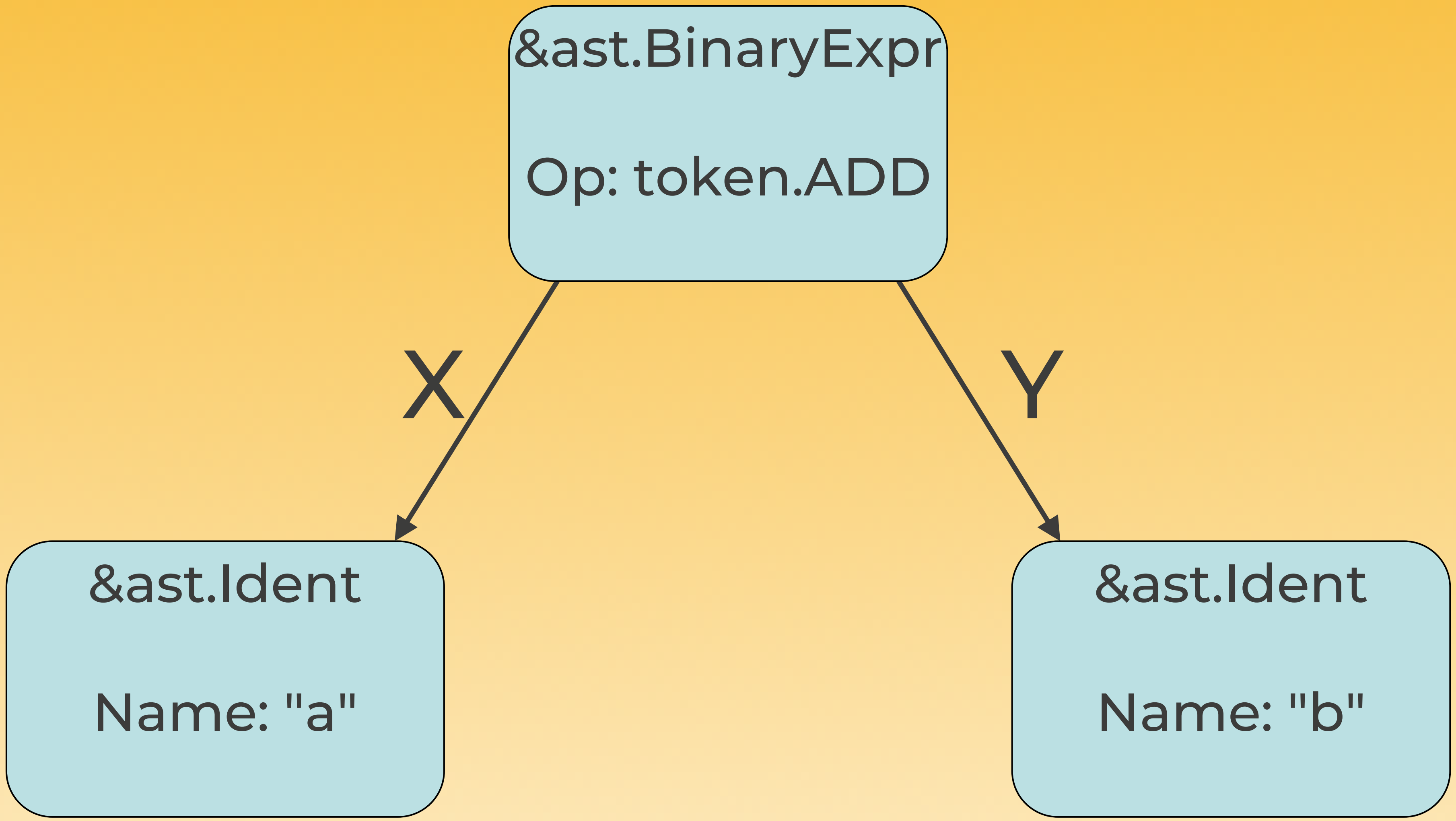
Name: "a"

&ast.Ident

Name: "b"

Manipulating Go source code

Abstract Syntax Tree (AST): a+b



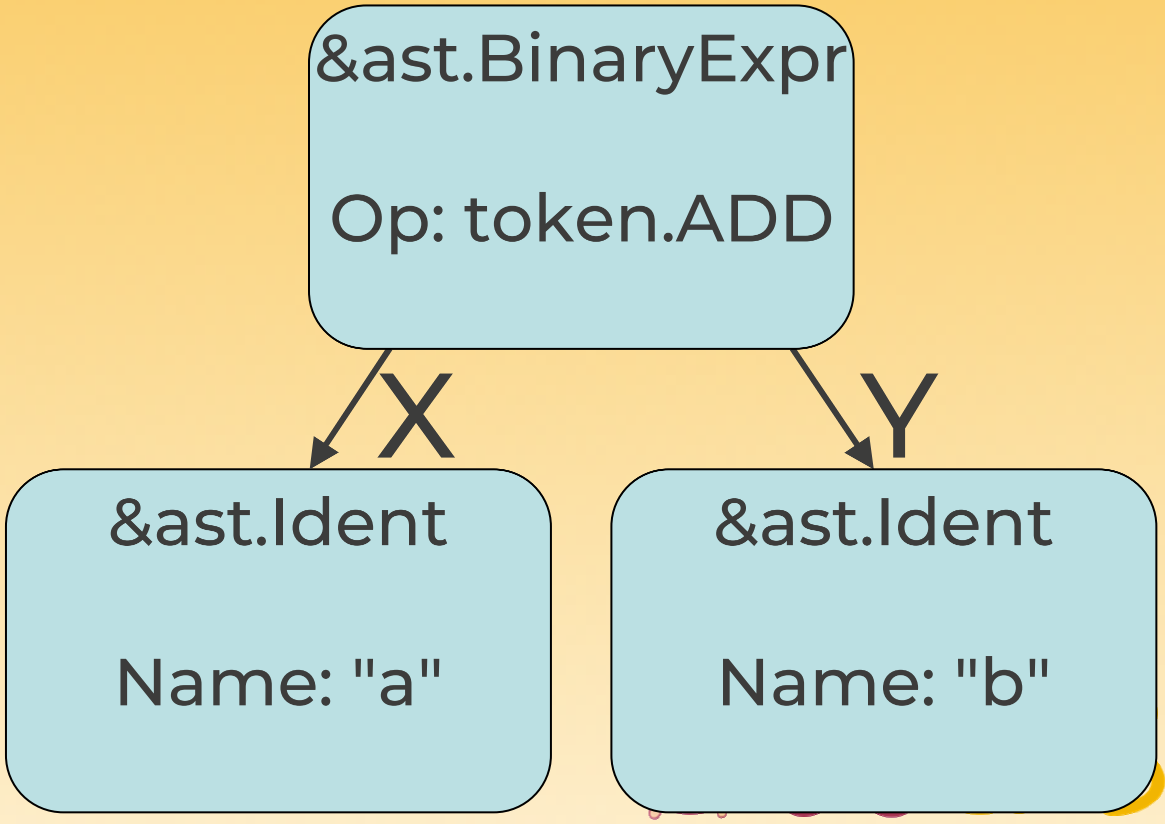
Manipulating Go source code

Abstract Syntax Tree (AST):

a+b

```
import ( "go/ast"; "go/token" )

add := &ast.BinaryExpr{
    X:    &ast.Ident{Name: "a"},
    Op: token.ADD,
    Y:    &ast.Ident{Name: “b"},
}
```



&ast.BinaryExpr

Op: token.ADD

X

Y

&ast.Ident

Name: "a"

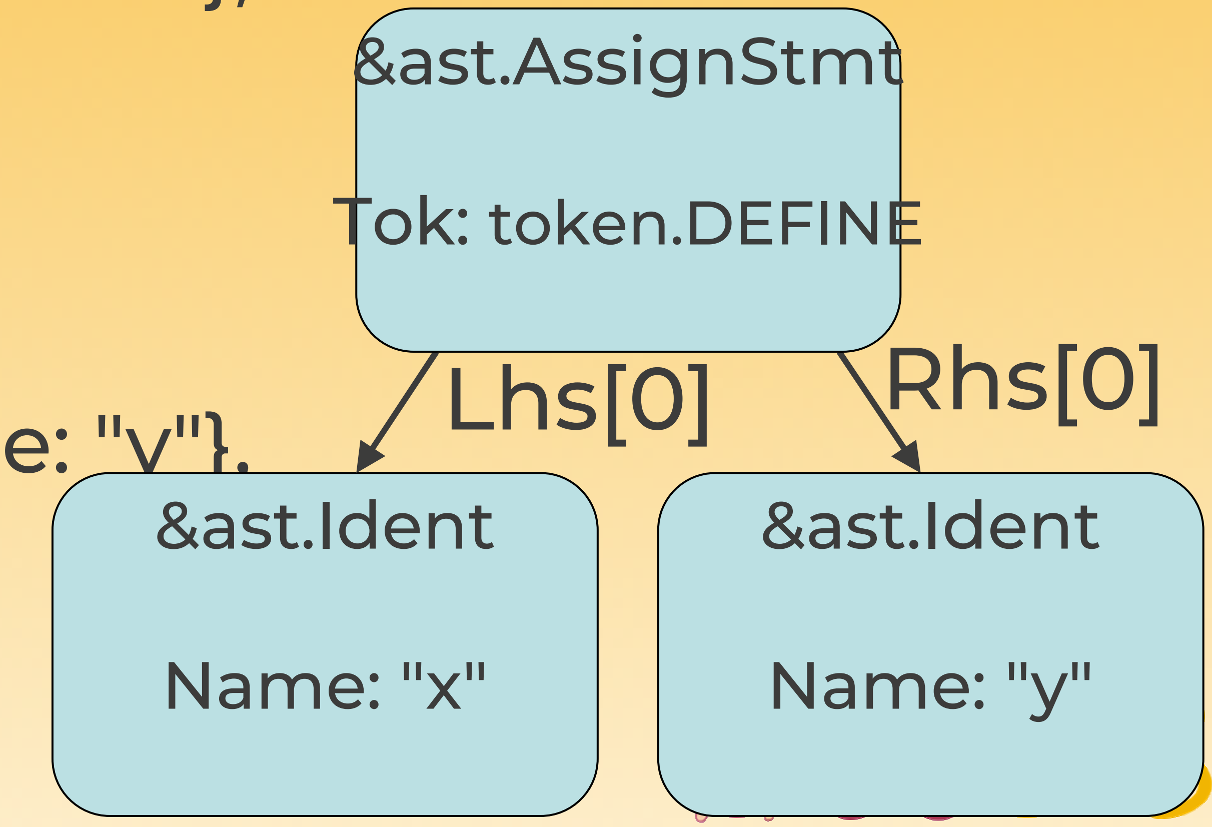
&ast.Ident

Name: "b"

Manipulating Go source code

Abstract Syntax Tree (AST):   x := y

```
assign := &ast.AssignStmt{
    Lhs:   []ast.Expr{
                &ast.Ident{Name: "x"},
           },
    Tok:   token.DEFINE,
    Rhs:   []ast.Expr{
                &ast.Ident{Name: "y"},
           },
}
```



&ast.AssignStmt

Tok: token.DEFINE

Lhs[0]

Rhs[0]

&ast.Ident

Name: "x"

&ast.Ident

Name: "y"

Pros & cons

Good to have a standardized AST representation of Go source in the standard library

But too verbose: creating AST structs manually gets cumbersome, error-prone and unreadable pretty fast

Better solutions?

- go/parser
- github.com/dave/jennifer
- github.com/bouk/gonerics
- github.com/cosmos72/gomacro
- ...

go/parser

```
import "go/parser"
```

```
add, err := parser.ParseExpr("a+b")
assign, err := // not supported!
```

Can only parse an expression, a whole file or a whole directory.

No API to parse a statement or declaration.

dave/jennifer

```
import . "github.com/dave/jennifer/jen"
add := Id("a").Op("+").Id("b")
assign := Id("x").Op(":=").Id("y")
```

Already better, but...

- custom representation, not go/ast
- still not obvious to use: blocks? start from which token?

bouk/gonerics

Aimed at generics, not code generation

cosmos72/gomacro

```
add := ~'{a+b}
assign := ~'{x:=y}
```

- usual Go syntax, tiny overhead
- builds go/ast structs
- very readable

- disadvantages: language extension

cosmos72/gomacro

Go interpreter:

- REPL with full Go standard library
- 3rd party imports compiled and loaded dynamically (Linux, Mac OS X)
- almost full Go language support
- debugger
- generics – currently styled after C++ templates, not Go 2 proposal
- code generation and macros

gomacro example (1)

```
$ go get github.com/cosmos72/gomacro
$ gomacro
// greeting...

gomacro> 1+2
{int 3}  // untyped.Lit
gomacro> 1<<100
{int 1267650600228229401496703205376}
```

gomacro example (2)

```
gomacro> func fib(n int) int {
.   .   .   .    if n <= 2 { return 1 }
.   .   .   .    return fib(n-1) + fib(n-2)
.   .   .   .   }

gomacro> fib(30)
832040  // int

gomacro> fib
0xc000236ac0  // func(int) int
```

quoting example (1)

```
gomacro> add := ~'{a+b} // ask nicely for AST
gomacro> add
a + b  // *go/ast.BinaryExpr
gomacro> :inspect add
add     = a + b  // *go/ast.BinaryExpr
        0.  X          = {Name:a ...} // ast.Expr
        1.  OpPos      = 283 // token.Pos
        2.  Op         = + // token.Token
        3.  Y          = {Name:b ...} // ast.Expr
```

quoting example (2)

```
gomacro> assign := ~'{x:=y}
gomacro> assign
x := y  // *go/ast.AssignStmt
gomacro> :inspect assign
assign     = x := y  // *go/ast.AssignStmt
        0.  Lhs         = {x} // []ast.Expr
        1.  TokPos  = 311 // token.Pos
        2.  Tok         = := // token.Token
        3.  Rhs         = {y} // []ast.Expr
```

quoting syntax (1)

~'	~quote
~"	~quasiquote
~,	~unquote
~@,	~unquote_splice

```
gomacro> add := ~'{a+b}
gomacro> mul := ~"{3 * ~,add}
gomacro> mul
3 * (a + b)  // *go/ast.BinaryExpr
```

quoting syntax (2)

```
gomacro> list := ~'{1; 2; 3}
gomacro> c := ~"{case ~,@list: return}
gomacro> c
case 1, 2, 3:
        return          // *go/ast.CaseClause
```

downside:

nested quasiquotes and unquotes are notoriously tricky to write and understand

hello world

```
gomacro> h := ~'{
   import "fmt"
   ~func main() {
        fmt.Println("hello, world!")
   }
}
```

nice, but why ~func ?

new keywords

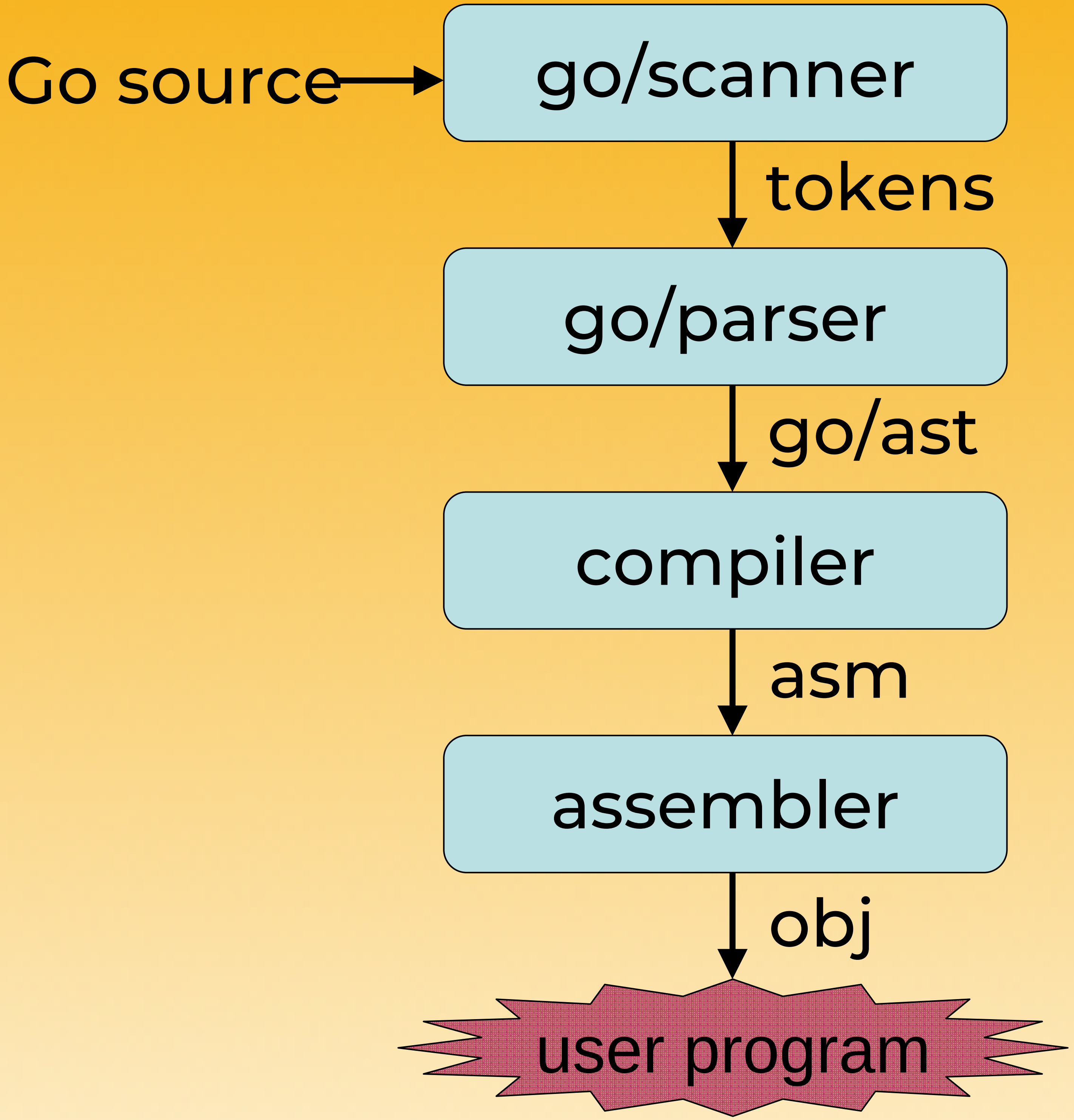
~func	non-top-level func/method
~lambda	top-level closure
~typecase	type case outside type switch
~macro	func executed at compile-time
~quote	
~quasiquote	
~unquote	
~unquote_splice	

is it enough ?

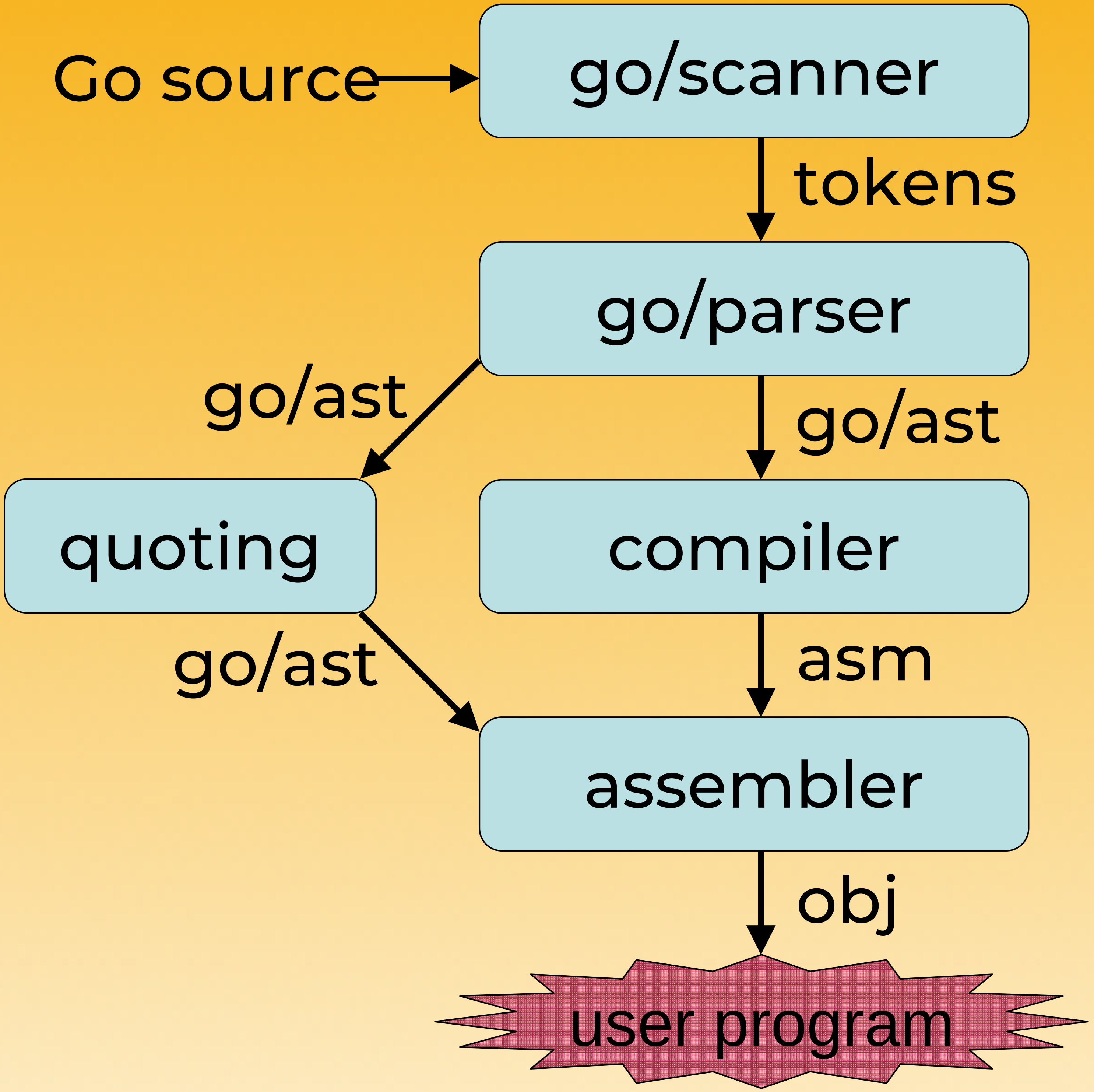
- easy to get AST from compiler and manipulate it at runtime
- AST could be manually dumped to Go source file

- what about giving *back* AST to the compiler?
- welcome to macros!

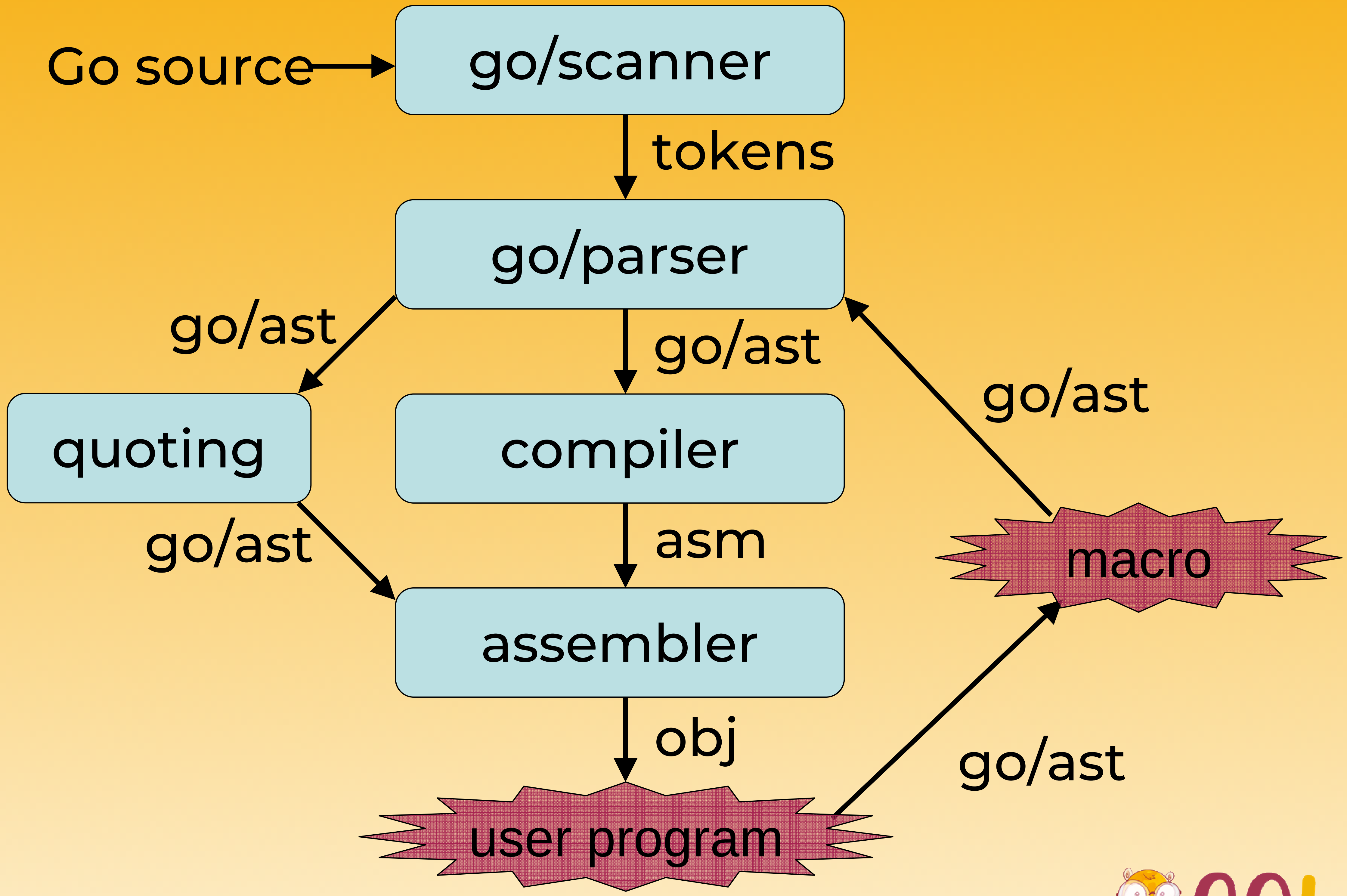
a bit of compiler architecture



a bit of compiler architecture



a bit of compiler architecture



macros

A macro is:

- a normal function
- executed at compile-time
  i.e. while compiler runs
- its input is source code AST
- its output is transformed AST to be compiled

macro example (1)

```
import "go/ast"
macro add(a, b ast.Node) ast.Node {
   return ~"{~,a + ~,b}
}
add; 1; 2
{int 3} // Untyped.Lit

add; "x"; "y"
{string "xy"} // Untyped.Lit
```

macro example (2)

```
import "go/ast"
macro makefib(typ ast.Node) ast.Node {
    return ~"{
        ~func fib(n ~,typ) ~,typ {
            if n <= 2 {
                return 1
            }
            return fib(n-1) + fib(n-2)
        }
    }
}
```

macro example (3)

```
makefib; int
fib(30)
832040 // int

makefib; uint64
fib(30)
832040 // uint64
```

debugging macros

```
MacroExpand1(~'{makefib; int})
// ...

MacroExpand(~'{makefib; uint64})
// ...
```

can also use println(), fmt.Printf() ...
or gomacro builtin debugger

macros as preprocessor

split macro expansion from evaluation

```
gomacro -f -m -w file.gomacro
```

-m means macroexpand-only

- code prefixed with : is evaluated
- other code copied as-is

Thanks!

Q&A

Contacts:

https://github.com/cosmos72/gomacro

massimiliano.ghilardi@gmail.com

gomacro statistics

two interpreters:

- classic – 6k LOC hand written
  directly interprets AST
  ~2000 times slower than compiled Go

- fast – 120k LOC
  81% generated with macros (!)
  "compiles" AST to tree of closures
  ~10-100 times slower than compiled Go