Creare un interprete in Go

Massimiliano Ghilardi, MBI s.r.l.

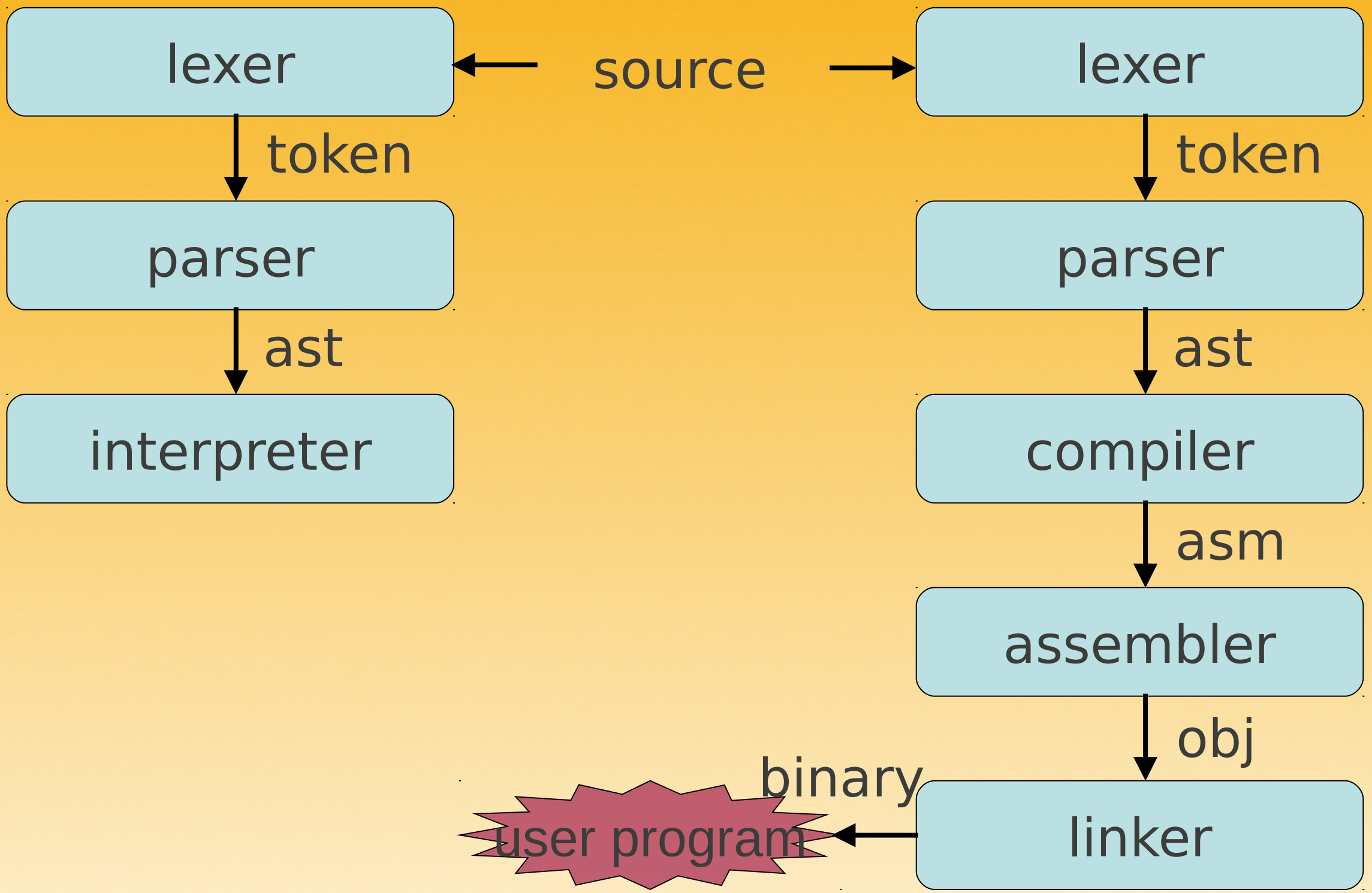
Linux Day 2019, Pisa

2019-10-26

Argomenti

- Interprete vs compilatore
- Lexer e parser
- Cos’è l’AST
- Interpretare l’AST
- Cosa manca?
- Conclusioni
- Extra: interpreti più veloci?
- Q & A

Interprete vs compilatore



Lexer (scanner)

```
package main

func main() {

}
```


```
PACKAGE
IDENT "main"
FUNC
IDENT "main"
LPAREN          // (
RPAREN          // )
LBRACE          // {
RBRACE          // }
```

Un po' laborioso da implementare a mano...

Per i sorgenti Go:

```
import "go/scanner"
import "go/token"
```

Lexer (scanner)

```
import "go/scanner"
import "go/token"

type Mode          uint                 // ScanComments
type ErrorHandler  func(/*...*/)
type Scanner       struct { /*...*/ }
```

```
func (*Scanner) Init(*token.File, []byte, ErrorHandler, Mode)

func (*Scanner) Scan() (token.Pos, token.Token, string)
```

token.Pos:	posizione
token.Token:	keyword, IDENT o costante letterale (INT...)
string:	nome identificatore o testo letterale

Parser

```
PACKAGE
IDENT "main"
FUNC
IDENT "main"
LPAREN          // (
RPAREN          // )
LBRACE          // {
RBRACE          // }
```

go/token

go/parser

go/ast

```
&ast.FuncDecl{
  Name:    &ast.Ident{"main"},
  Type:    &ast.FuncType{...},
  Body:    &ast.BlockStmt{...},
}
```

Laborioso da implementare a mano

Per i sorgenti Go:

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

Parser

```
import "go/parser"

func ParseExpr(x string)   (ast.Expr, error)
func ParseFile(/*...*/)    (*ast.File, error)
func ParseDir(/*...*/)     (map[string]*ast.Package, error)
```

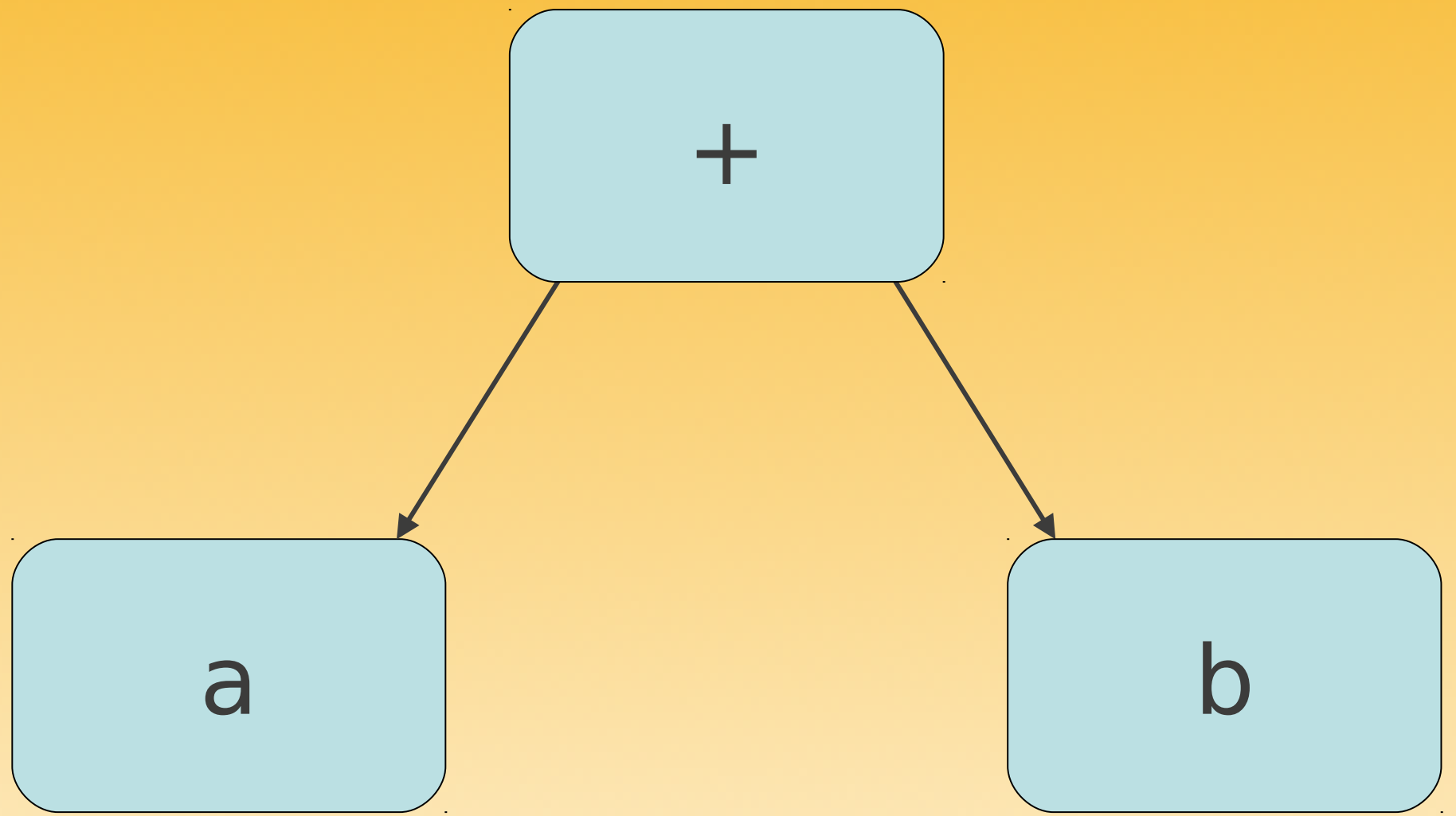
Mancano ParseStmt e ParseDecl, risolvibile:

```
ParseFile("package main; " + decl)

ParseFile("package main; func main() {" + stmt +
          "}")
```

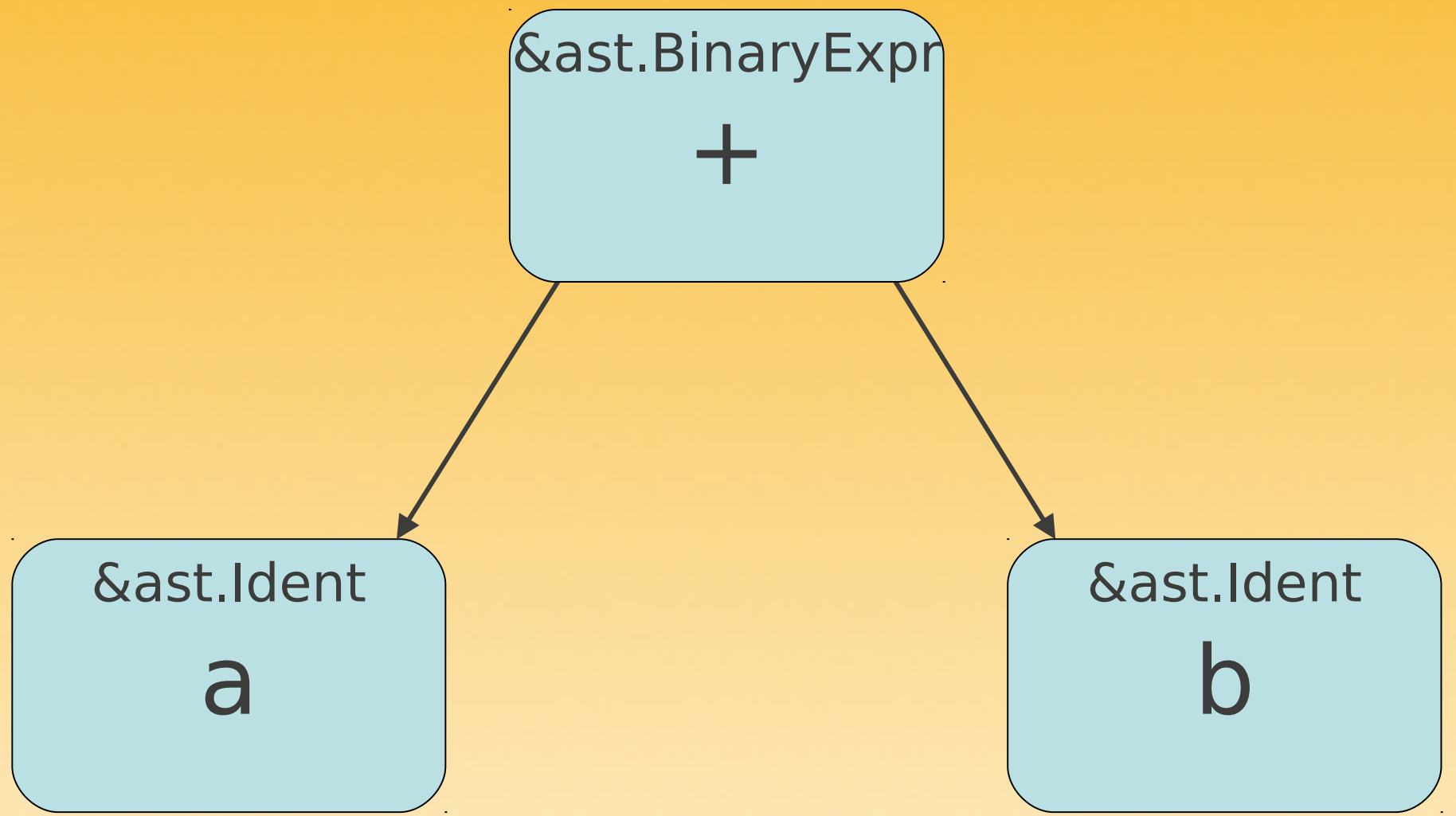
Cos’è l’AST

Abstract Syntax Tree (AST): a+b



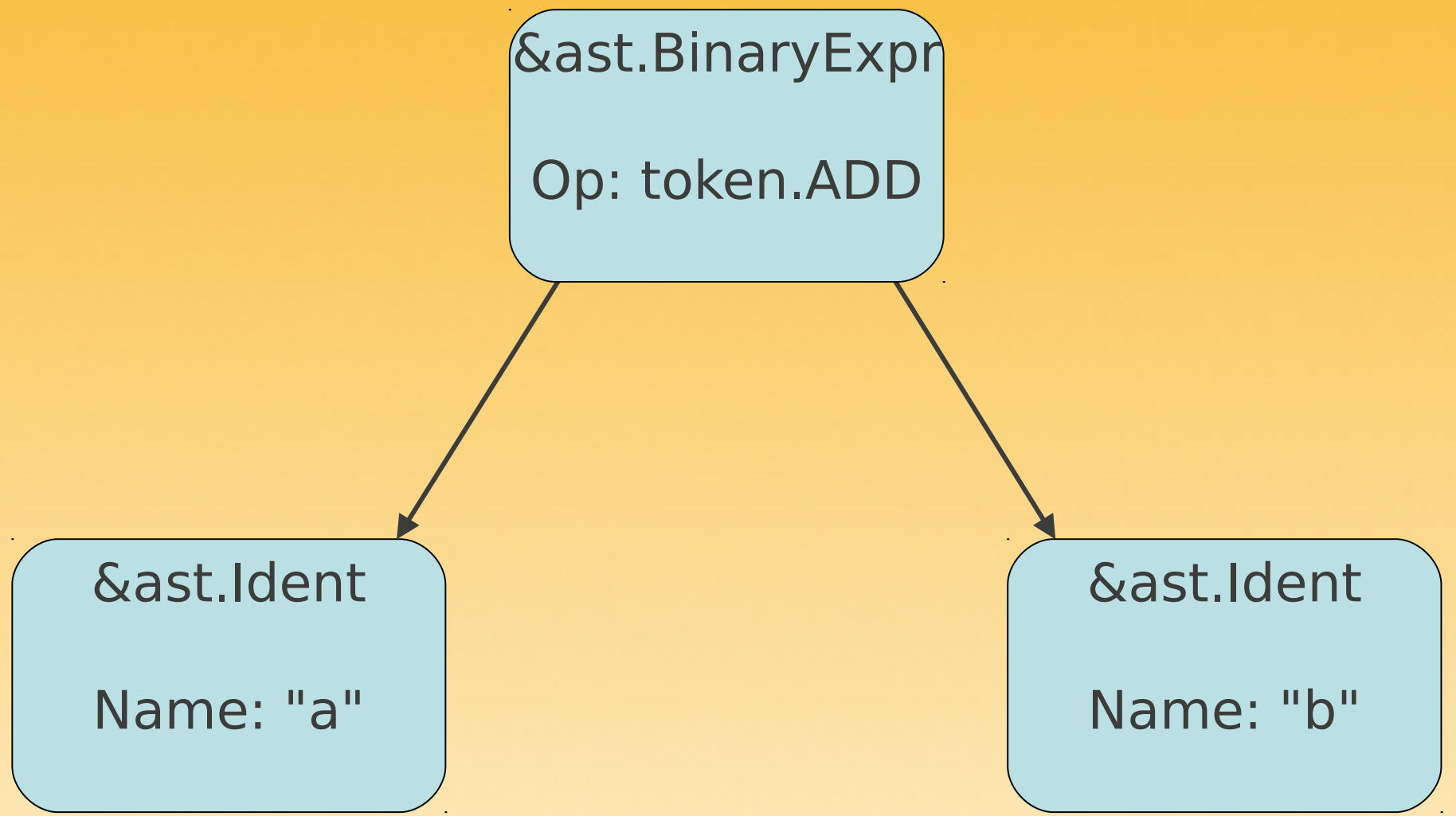
Cos’è l’AST

Abstract Syntax Tree (AST): a+b



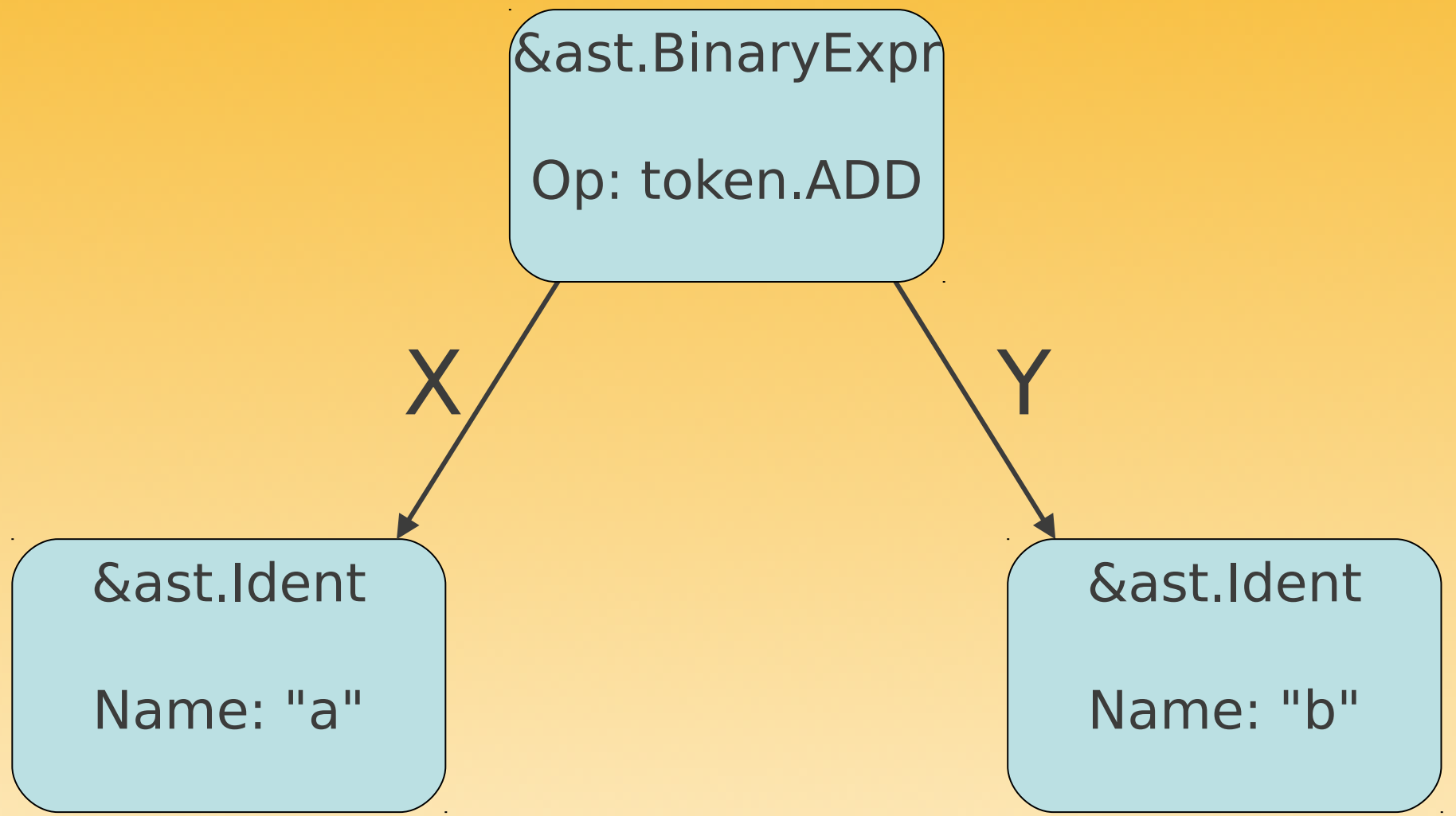
Cos’è l’AST

Abstract Syntax Tree (AST): a+b



Cos’è l’AST

Abstract Syntax Tree (AST): a+b

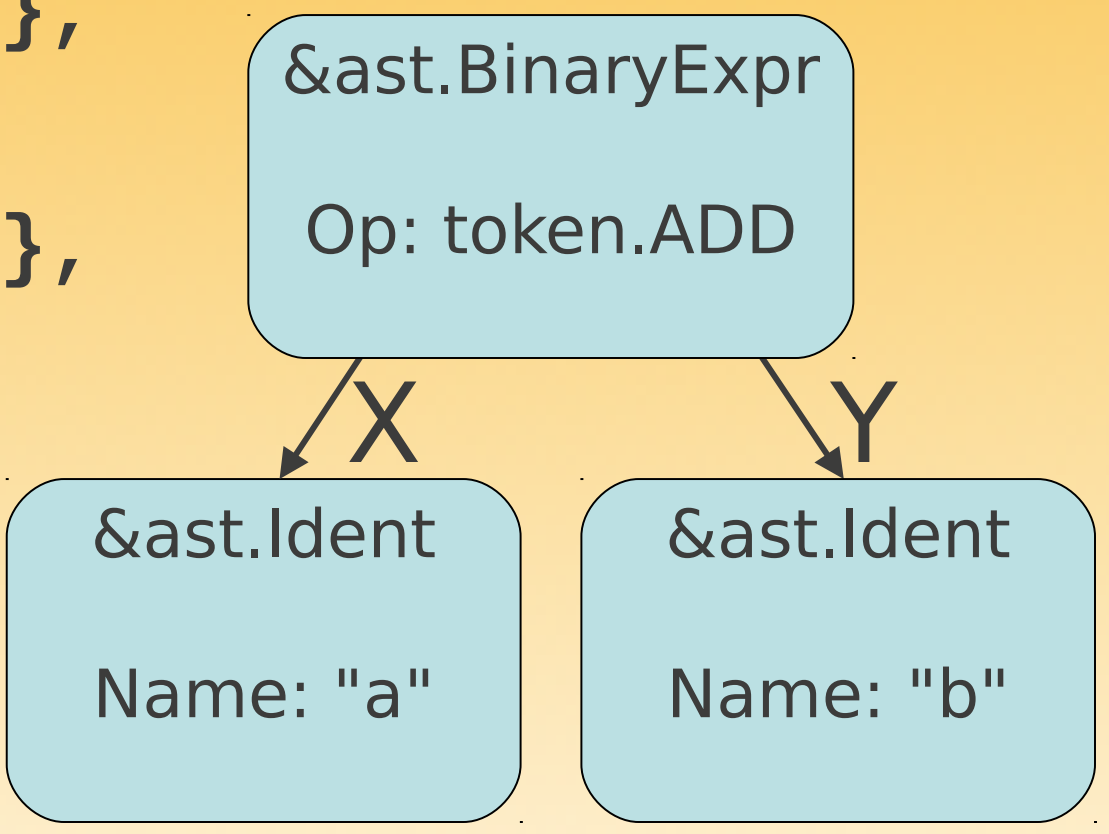


Cos’è l’AST

Abstract Syntax Tree (AST): a+b

Creato per noi dal parser

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

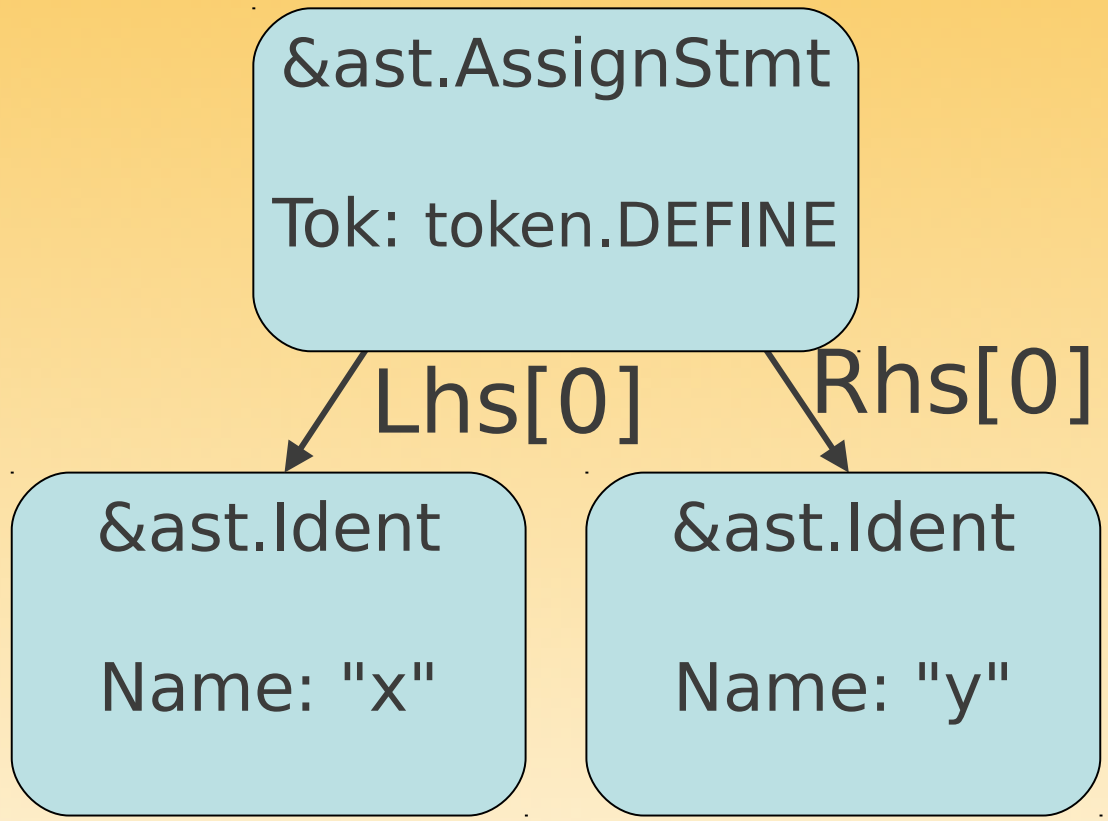


Cos’è l’AST

Abstract Syntax Tree (AST): x := y

Creato per noi dal parser

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



Interpretare l’AST

Interpretazione diretta:

- È la tecnica più semplice
- Ci aiuta reflect.Value
- Molto lento

Interpretare l’AST - Node

```
func (env *Env) EvalNode(node ast.Node) []r.Value {
    switch node := node.(type) {
        case *ast.File:
            env.evalFile(node)
        case ast.Decl:
            env.evalDecl(node)
        case ast.Stmt:
            env.evalStmt(node)
        case ast.Expr:
            return env.evalExpr(node)
        default:
            panic("unsupported ast.Node")
    }
    return nil
}
```

Interpretare l’AST - Expr

```
func (env *Env) evalExpr(node ast.Expr) []r.Value {
    switch node := node.(type) {
        case *ast.Ident:
            return env.evalIdent(node)
        case *ast.BinaryExpr:
            return env.evalBinaryExpr(node)
        case *ast.UnaryExpr:
            return env.evalUnaryExpr(node)
        // ...
    }
    return nil
}
func (env *Env) evalExpr1(node ast.Expr) r.Value {
    v := env.evalExpr(node)
    if len(v) != 1 { panic("expecting a single value") }
    return v[0]
}
```

Interpretare l’AST - BinaryExpr

```
func (env *Env) evalBinaryExpr(
        node *ast.BinaryExpr) []r.Value {
    // TODO: token.AND, token.OR sometimes skip y
    x := env.evalExpr1(node.X)
    y := env.evalExpr1(node.Y)
    switch node.Op {
        case token.ADD:
            return env.add(x, y)
        case token.SUB:
            return env.sub(x, y)
        case token.MUL:
            return env.mul(x, y)
        // ...
    }
}
```

Interpretare l’AST - add

```
func (env *Env) add(x, y r.Value) []r.Value {
    if x.Type() != y.Type() {
        panic("operands of + must have the same type")
    }
    var ret interface{}
    switch x.Kind() {
        case r.Int, r.Int8, r.Int16, r.Int32, r.Int64:
            ret = x.Int() + y.Int()
        case r.Uint, r.Uint8, r.Uint16, r.Uint32 /*...*/ :
            ret = x.Uint() + y.Uint()
        case r.Float32, r.Float64:
            ret = x.Float() + y.Float()
        // ...
    }
    // Convert() truncates and supports named basic types
    return []r.Value{r.ValueOf(ret).Convert(x.Type())}
}
```

Interpretare l’AST - func

```
func (env *Env) evalFuncDecl(node *ast.FuncDecl) {
    // non banale
    /*...*/
    name := node.Name.Name
    env.Bind[name] = r.makeFunc(/*...*/)
}
```

```
func (env *Env) evalCallExpr(node *ast.CallExpr) []r.Value {
    // TODO: chiamata a funzioni variadiche
    funv := env.evalExpr1(node.Fun)
    argv := make([]r.Value, len(node.Args))
    for i, arg := range node.Args {
       argv[i] = env.evalExpr1(arg)
    }
    return funv.Call(argv) // reflect!
}
```

Interpretare l’AST – return...

```
func (env *Env) evalReturn(node *ast.ReturnStmt) {
    // non banale, deve saltare a fine funzione!
    // la soluzione più diretta è:
    panic(env.makeReturnValues(/*...*/))
    // richiede che evalFuncDecl chiami recover()
    // per fermare il panic e ottenere i valori
}


func (env *Env) evalBranchStmt(node *ast.BranchStmt) {
    // BREAK, CONTINUE, GOTO: non banale. usare panic?
    // FALLTHROUGH: restituire un valore speciale
}
```

Cosa manca?

- variabili locali e regole di scoping
- dichiarazione di tipi e variabili
- dichiarazione named types (ricorsivi), interfacce
- untyped constants, iota
- channels, send, receive
- statements: if, for, switch, select
- funzioni speciali: append, copy, make, new
- assegnamento singolo e multiplo
- keyword speciali: defer, recover
- funzioni, closures e metodi
- type inference, type checking
- goroutines
- import, unsafe

Cosa manca?

- variabili locali e regole di scoping
- dichiarazione di tipi e variabili
- dichiarazione named types (ricorsivi), interfacce
- untyped constants, iota
- channels, send, receive
- statements: if, for, switch, select
- funzioni speciali: append, copy, make, new
- assegnamento singolo e multiplo
- keyword speciali: defer, recover
- funzioni, closures e metodi
- type inference, type checking
- goroutines
- import, unsafe

Variabili locali e regole di scoping

```
type Env struct {
    Bind  map[string]r.Value
    Type  map[string]r.Type
    Outer *Env
    /*...*/
}
// usato ogni volta che troviamo { }
func newEnv(outer *Env) *Env {
    return &Env{
        make(map[string]r.Value),
        make(map[string]r.Type),
        outer}
}
```

Dichiarazione di variabili

```
func (env *Env) evalVar1(name string, t r.Type, v r.Value) {
    place := r.New(t).Elem() // settable t
    place.Set(v)
    env.Bind[name] = place  // not thread safe!
}
```

Dichiarazione di variabili

```
// var a, b, c int = foo(), bar(), baz()

func (env *Env) evalVar(node *ast.ValueSpec) {
    t := env.evalType(node.Type) // TODO nil => must infer
    init := make([]r.Value, len(node.Names))
    for i := range node.Names {
        if i < len(node.Values) {
            init[i] = env.evalExpr1(node.Values[i])
        } else {
            init[i] = r.Zero(t) // TODO var a,b,c = foo()
        }
    }
    // declare vars AFTER all init[i] = eval()
    for i, name := range node.Names {
        env.evalVar1(name, t, init[i])
    }
}
```

Conclusioni

Laborioso ma non troppo

Alcune lacune di "reflect"

Qualche difficoltà inaspettata:

- defer, recover
- goto

Quattro funzionalità molto difficili:

- nuovi named types ricorsivi
- nuove interfacce
- import
- unsafe

Velocità: ~2000 volte più lento del Go compilato

Dimensioni: ~6000 linee di codice

Extra: interpreti più veloci?

compilatore bytecode + interprete:
"il peggio di entrambi i mondi"

compilatore JIT (just-in-time):
non portabile, dipende dalla CPU
difficile quasi quanto un compilatore
impossibile in Go (no stack maps API)

tree-of-closures:
più facile delle alternative
~10-100 volte più lento del Go compilato

Extra: tree-of-closures

```
type Expr func(*Env) int // non reflect.Value

func (c *Comp) add(x, y Expr) Expr {
    return func(env *Env) int {
        return x(env) + y(env)
    }
}

type Env struct {
    // slice predimensionato:
    // veloce e thread-safe
    Int     []int
    Outer *Env
}
```

Extra: tree-of-closures

Difficoltà?

moltissime closure ripetitive:
- almeno una per tipo per ogni operazione fattibile **generando** il sorgente Go

Comp somiglia ad un compilatore:
- converte da AST ad albero di closure
- conta le variabili sullo stack
- valuta tipi e costanti
- calcola gli indirizzi di destinazione dei jump

Extra: statements

```
type Stmt func(*Env) (Stmt, *Env)

type Env struct {
    Int []int
    Exec    []Stmt  // predimensionato:
                    // break, continue, goto sono facili
    IP  int
    Outer   *Env
}
```

```
func (c *Comp) exprStmt(node *ExprStmt) Stmt {
    expr := /*...*/ // closure che valuta l'espressione
    return func(env *Env) (Stmt, *Env) {
        expr(env)
        env.IP++
        return env.Exec[env.IP], env
    }
}
```

Extra: statements

core dell'interprete

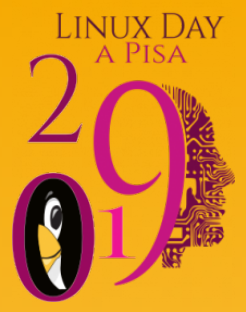
```
func (env *Env) run() {
    stmt := env.Exec[0]
    env.IP = 0
    for stmt != nil {
        // unroll for speed
        stmt, env = stmt(env)
    }
}
```

Extra: tree-of-closures

Difficoltà?

moltissime closure ripetitive:
- almeno una per tipo per ogni operazione fattibile **generando** il sorgente Go

Comp somiglia ad un compilatore:
- converte da AST ad albero di closure
- conta le variabili sullo stack
- valuta tipi e costanti
- crea array di statements e jump relativi



Grazie!

Q&A

Contatti:

massimiliano.ghilardi@gmail.com

https://github.com/cosmos72/gomacro

statistiche di gomacro

due interpreti:

- classic – 6k linee, scritte a mano
  interpreta direttamente l’AST
  ~2000 volte più lento del Go compilato

- fast – 115k linee
  80% **generate** usando macro
  "compila" l’AST in un albero di closure
  ~10-100 volte più lento del Go compilato