The Visitor Pattern in Rust

Sagar Kale

About me

- Name: Sagar Kale
- Theoretical computer scientist
- Moved to industry last year
- Coming from OO languages (Java, C++) to Rust

A new language called Lox

```
print 42;

fun fib(n) {
    if (n == 0 or n == 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

print fib(10);
```

Most slides based on book Crafting Interpreters by Robert Nystrom.

Crafting Interpreters

—Robert Nystrom

Lox is a language designed for this AWESOME, extremely well-written, and freely available book.

Lox grammar

```
program         → declaration* EOF ;
declaration     → classDecl
                | funDecl
                | varDecl
                | statement ;
statement       → exprStmt | forStmt | ifStmt | printStmt | returnStmt | whileStmt | block ;
```

`Decl`s and `Stmt`s skipped.

Lox grammar (expression)

```
expression     → assignment ;

assignment     → ( call "." )? IDENTIFIER "=" assignment | logic_or ;
logic_or       → logic_and ( "or" logic_and )* ;
logic_and      → equality ( "and" equality )* ;
equality       → comparison ( ( "!=" | "==" ) comparison )* ;
comparison     → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term           → factor ( ( "-" | "+" ) factor )* ;
factor         → unary ( ( "/" | "*" ) unary )* ;
unary          → ( "!" | "-" ) unary | call ;
call           → primary ( "(" arguments? ")" | "." IDENTIFIER )* ;
primary        → "true" | "false" | "nil" | "this"
                | NUMBER | STRING | IDENTIFIER | "(" expression ")"
                | "super" "." IDENTIFIER ;
```

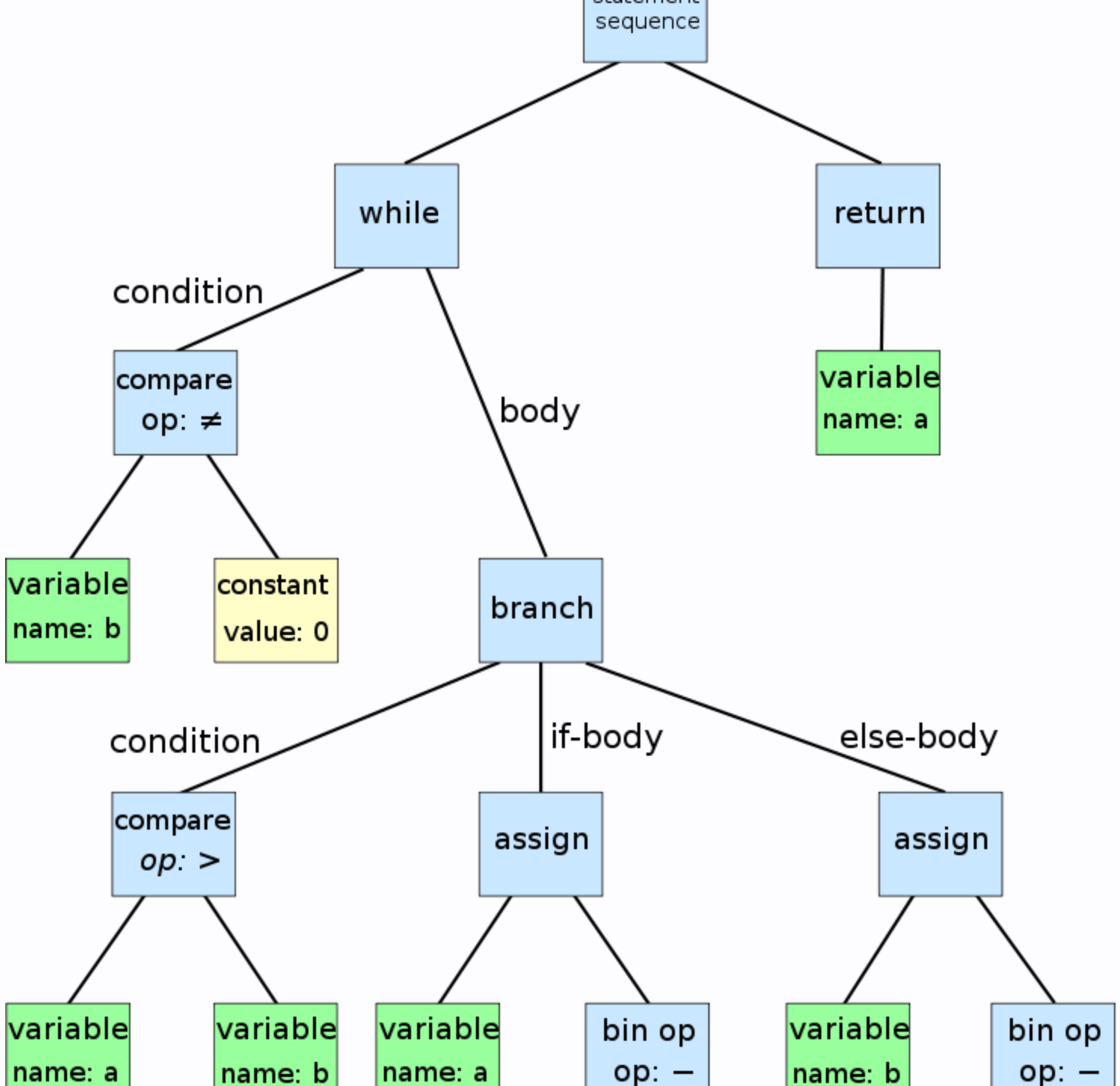
Utility and lexical rules skipped.

Abstract Syntax Tree (AST)

for Euclid's algorithm

```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

This and the next slide from Wikipedia.


statement sequence
while
return
condition
body
compare op: ≠
variable name: a
variable name: b
constant value: 0
branch
condition
if-body
else-body
compare op: >
assign
assign
variable name: a
variable name: b
variable name: a
bin op op: −
variable name: b
bin op op: −

The Expression Problem

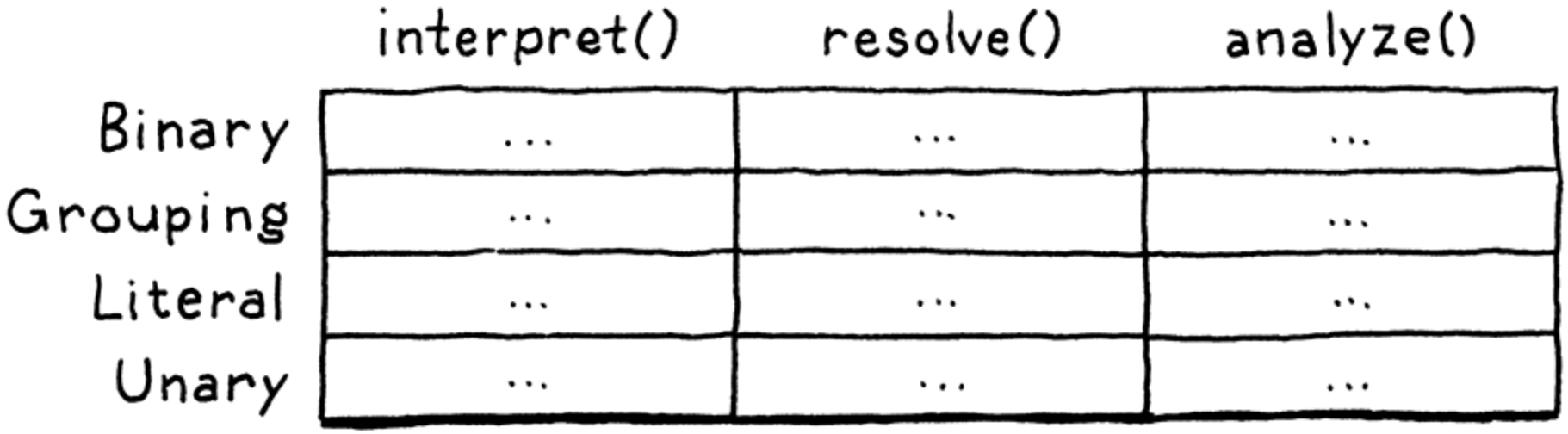


Diagram taken from Crafting Interpreters by Robert Nystrom.

The Visitor Pattern

> The Visitor pattern is the most widely misunderstood pattern in all of Design Patterns.
>
> –Robert Nystrom.

```
abstract class Pastry {}

class Beignet extends Pastry {}

class Cruller extends Pastry {}
```

```
interface PastryVisitor {
  void visitBeignet(Beignet beignet);
  void visitCruller(Cruller cruller);
}
```

The Visitor Pattern (contd.)

```
abstract class Pastry {
  abstract void accept(PastryVisitor visitor);
}

class Beignet extends Pastry {
  @Override
  void accept(PastryVisitor visitor) {
    visitor.visitBeignet(this);
  }
}

class Cruller extends Pastry {
  @Override
  void accept(PastryVisitor visitor) {
    visitor.visitCruller(this);
  }
}
```

The Visitor Pattern (contd.)

Just call `Pastry.accept()`.

Implement as many visitors as you want

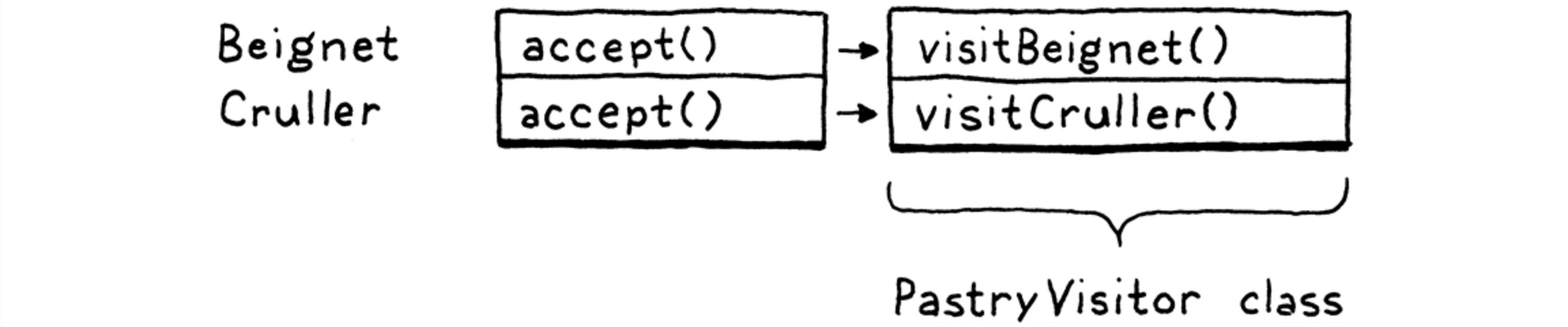


Diagram from Crafting Interpreters by Robert Nystrom.

The Visitor for `Expr`s

```
abstract class Expr {
  abstract <R> R accept(Visitor<R> visitor);
}

interface Visitor<R> {
    R visitBinaryExpr(Binary expr);
    R visitGroupingExpr(Grouping expr);
    R visitLiteralExpr(Literal expr);
    R visitUnaryExpr(Unary expr);
  }
```

A wrong Visitor in Rust

```
trait Expr {
    fn accept(&self, visitor: &dyn Visitor) -> VisitorReturnResult;
}

trait Visitor {
    fn visit_binary_expr(&self, expr: &Binary) -> VisitorReturnResult;
    fn visit_grouping_expr(&self, expr: &Grouping) -> VisitorReturnResult;
    fn visit_literalexpr_expr(&self, expr: &LiteralExpr) -> VisitorReturnResult;
    fn visit_unary_expr(&self, expr: &Unary) -> VisitorReturnResult;
}

struct Binary {
    left: Box<dyn Expr>,
    operator: Token,
    right: Box<dyn Expr>,
}

impl Expr for Binary {
    fn accept(&self, visitor: &dyn Visitor) -> VisitorReturnResult {
        visitor.visit_binary_expr(&self)
    }
}
```

Using `enum`s

```
enum Expr {
    BinaryExpr(Box<Binary>),
    GroupingExpr(Box<Grouping>),
    LiteralExprExpr(Box<LiteralExpr>),
    UnaryExpr(Box<Unary>),
}

impl Expr {
    fn accept<R>(&self, visitor: &mut dyn Visitor<R>) -> R {
        match self {
            Expr::BinaryExpr(expr) => visitor.visit_binary_expr(expr),
            Expr::GroupingExpr(expr) => visitor.visit_grouping_expr(expr),
            Expr::LiteralExprExpr(expr) => visitor.visit_literalexpr_expr(expr),
            Expr::UnaryExpr(expr) => visitor.visit_unary_expr(expr),
        }
    }
}
trait Visitor<R> {
    fn visit_binary_expr(&mut self, expr: &Binary) -> R;
    fn visit_grouping_expr(&mut self, expr: &Grouping) -> R;
    fn visit_literalexpr_expr(&mut self, expr: &LiteralExpr) -> R;
    fn visit_unary_expr(&mut self, expr: &Unary) -> R;
}

impl expr::Visitor<ExprVisitorResult> for Interpreter {...}
```

Much ado about nothing

```
impl Interpreter {
    fn visit_expr(&mut self, e: &Expr) -> ExprVisitorResult {
        match *e {
            Expr::BinaryExpr(expr) => ...,
            Expr::GroupingExpr(expr) => ...,
            Expr::LiteralExprExpr(expr) => ...,
            Expr::UnaryExpr(expr) => ...,
        }
    }
}
```

But the previous way makes it more explicit that we are using a visitor pattern.

That's all!

- Crafting Interpreters
- The visitor pattern
- `enum` : answer to the ultimate question in Rust