

The Visitor Pattern in Rust

Sagar Kale

Slide # 5, 6, 9, 10, 11, 12, 13 taken from the book
Crafting Interpreters, therefore © Robert Nystrom.
Copyright and licensing information on the next slide.

Copyright and licensing information

Slide # 5, 6, 9, 10, 11, 12, 13 taken from the book Crafting Interpreters and accompanying materials, therefore © Robert Nystrom.

See <https://github.com/munificent/craftinginterpreters/blob/master/LICENSE>

Material from the above book on slide # 9, 10, 11, and 12 is licenced under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) license <https://creativecommons.org/licenses/by-nc-nd/4.0/> and is being used under the fair use clause in that license.

Material from the repository for the book that appears on slide # 5, 6, and 13 is already licensed under the MIT license, hence it can be sublicensed.

Slide # 7 and 8 are from a Wikipedia page (https://en.wikipedia.org/wiki/Abstract_syntax_tree) that uses Creative Commons Attribution-ShareAlike License 4.0 <https://creativecommons.org/licenses/by-sa/4.0/> , hence it can be sublicensed under the same license.

In summary, all slides except # 9, 10, 11, and 12 are licensed under Creative Commons Attribution-ShareAlike License 4.0 <https://creativecommons.org/licenses/by-sa/4.0/> by the author.

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);
```

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 ;
```

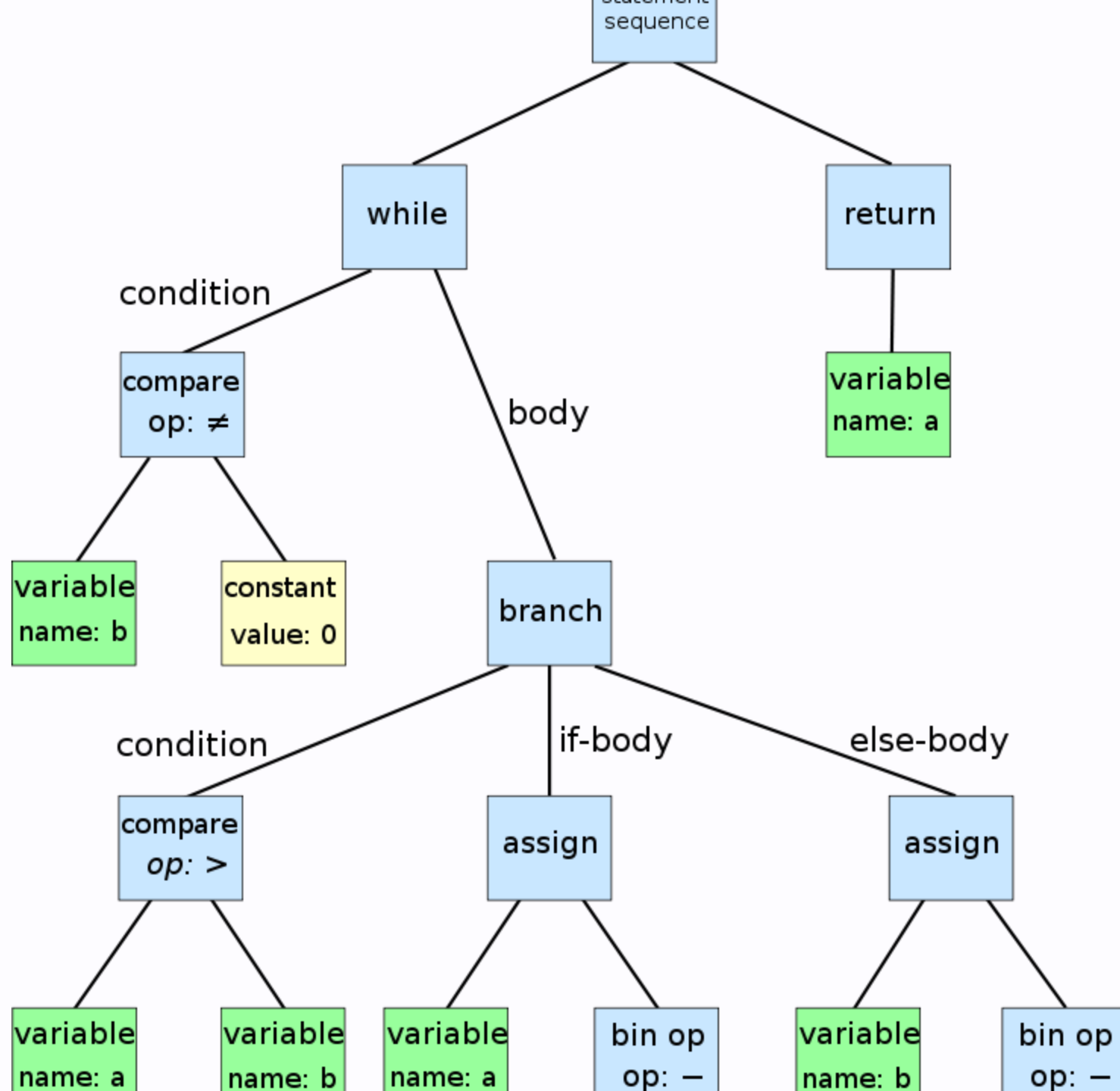
Utility and lexical rules skipped.

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.



The Expression Problem

	interpret()	resolve()	analyze()
Binary
Grouping
Literal
Unary

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);  
}
```

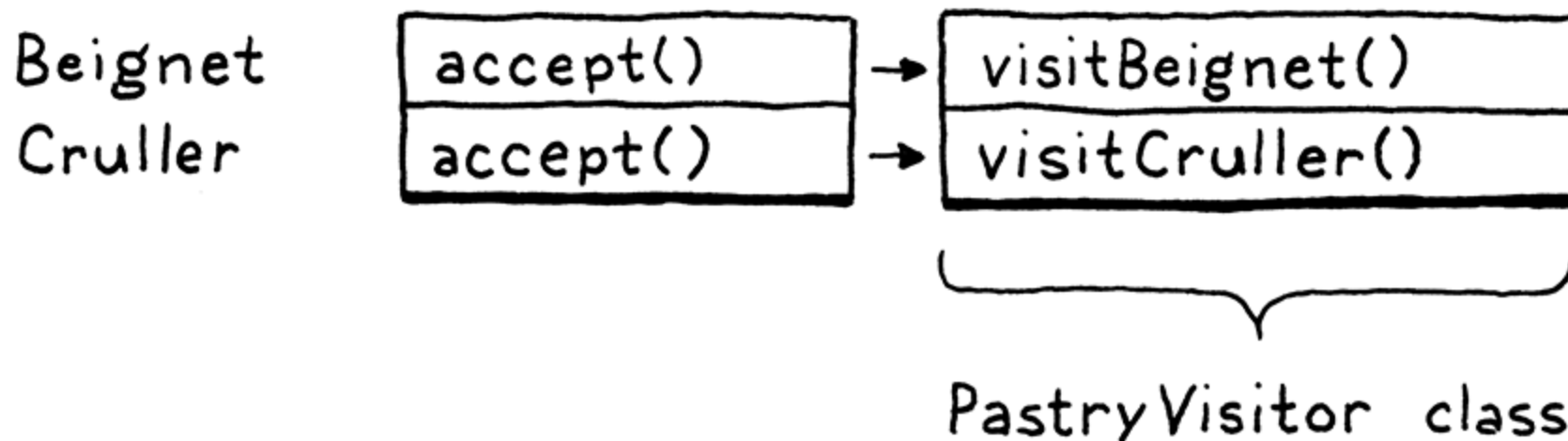
The Visitor Pattern (contd.)

```
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



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_literal_expr_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 `enums`

```
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