YARTBML

Yet Another Re-implementation of Thorsten Ball’s Monkey Language

By: Dinesh Umasankar, Joseph Porrino, Katherine Banis, Paul Jensen

What is YARTBML?

Minimally Functional-Paradigm Inspired
Language (Based on SMoL)

- 🛠️ Built on the foundation provided by Thorsten Ball’s: "Writing an Interpreter in Go"
- 💭 Inspired by the many forks of this foundation to provide the best feature set and experience
- 👩‍💻 Focuses on the developer experience for building general-purpose applications
- 🍕 Learnable in a lunch break

Data Types

- Integers: Whole numbers without a decimal component, e.g., `42`, `-7`.
- Booleans: Logical type representing `true` or `false`.
- Strings: A sequence of characters enclosed in double quotes, e.g., `"YARTBML is awesome!"`.
- Arrays: A list of elements, e.g., `[1, 2, 3, 4, "hello", true]`.
- Hashmaps: Key-value pairs, e.g., `{"name": "YARTBML", "isCool": true}`.

Data Types In Action

```
// Our team in an array
let team = [dinesh, joseph, katherine, paul];
let leader = team[0] // {"name": "Dinesh Umasankar", classification: "Senior"}
```

Functions

- First-Class Citizens
- Functions are a value-type

- Can be assigned to variables
- Passed as arguments
- Returned from other functions

```
let greet = fn(name) { return "Hello, " + name + "!"; };
let message = greet("World");
puts(message); // -> "Hello, World!"
```

Operators

Traditional Arithmetic Operators w/ Precedence

- Equality-Expression: `==` or `!=`
- Comparative Expression: `<` or `>`
- Additive-Expression: `+` or `-`
- Multiplicative-Expression: `*` or `/`
- Prefix-Expression: `-` or `!`

Project Components

Lexer

- Purpose is to tokenize text so parser can create an AST
- A token is a struct that holds a type and literal

```
type Token struct {
  Type    TokenType
  Literal string
}
```

- Lexer increments over each char in input string
- Tokenizes: Operators, delimiters, identifiers, keywords, and numbers

```
>> let x = 5
{Type:LET Literal:let}
{Type:IDENT Literal:x}
{Type:= Literal:=}
{Type:INT Literal:5}
>>
```

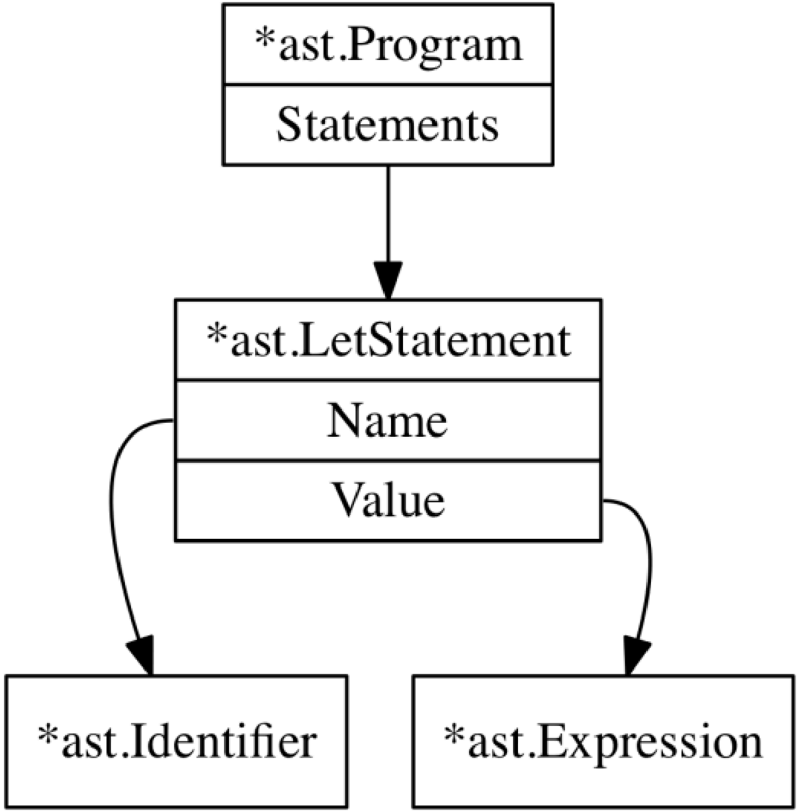
Parser

- Pratt parsing
- The image represents `let x = 5` as an AST
- Input are tokens from lexer
- Tokens get parsed and nodes are created

```go
type LetStatement struct {
  Token token.Token // token.LET token
  Name  *Identifier
  Value Expression
}
```

- AST is built by appending nodes to a list

```go
stmt := p.parseStatement()
if stmt != nil {
  program.Statements = append(program.Statements, stmt)
}
p.nextToken()
```



Evaluator

- Tree walks AST
- Start from root of AST and recurisvely evaluate each node
- Values are represented as objects to be passed through evaluator
- Environment holds identifier bindings

```
func Eval(node ast.Node, env *object.Environment) object.Object {
  switch node := node.(type) {
  // Statements
  case *ast.Program:
    return evalProgram(node, env)

  case *ast.ExpressionStatement:
    return Eval(node.Expression, env)

  case *ast.IntegerLiteral:
    return &object.Integer{Value: node.Value}
```

REPL

- Read Eval Print Loop
- Reads input from terminal
- Each statement goes through the lexer, parser, evaluator, than loops

REPL

```
scanner := bufio.NewScanner(in)
env := object.NewEnvironment()
for {
  fmt.Fprintf(out, PROMPT)
  scanned := scanner.Scan()
  if !scanned {
    return
  }
  line := scanner.Text()
  l := lexer.New(line)
  p := parser.New(l)
  program := p.ParseProgram()
  if len(p.Errors()) != 0 {
    printParserErrors(out, p.Errors())
    continue
  }
  evaluated := evaluator.Eval(program, env)
  if evaluated != nil {
    io.WriteString(out, evaluated.Inspect())
    io.WriteString(out, "\n")
  }
```

Syntax Highligher

- VSCode highlights code based on predefined rules
- Wrote regular expressions to match highlighting within a TextMate grammar to match our language
- Created a VSIX Extension

Memory Management

Handled by Go Language

- Go’s Runtime is statically linked into the interpreter binary, which contains a Garbage Collector.
- Interpreter is a binary file compiled to a specific machine architecture.

Running the program

- Clone repo to local machine
- Open folder in VSCode or other IDE
- Ensure your in the root directory of the project
- Run the following commands

```
cd internal
go run main.go
```

- REPL will start running

```
Hello JOEYS-PC\gympr! This is the YARTBML programming language!
Feel free to type in commands
>>
```

- Enter YARTBML code

Thank You