On ...

Stefanie

May 17, 2021

Abstract

Some thoughts on the book *Writing an Interpreter in Go* (Version 1.7) by Thorsten Ball, the Monkey Programming language and its implementation. So far, only the part of the interpreter implemented in chapter 3 is taken into account.

Contents

1 On Evaluating to Unorthodox Values

1.1 Statements vs Expressions

During the discussion of the parser, the author gives the following explanation for the difference between statements and expressions:

> Expressions produce values, statements don't.[1]

At least for the Monkey Programming language, this is not true, since there are statements in Monkey that produce values and expressions that don't (A part of Monkey is considered here not to produce a value if it evaluates to `nil` and not to a Monkey object.):

```
1 + 2        // ExprS
return 5     // RetS
if(true){5}  // ExprS
```

(a) Statements that produce values

```
let a = 5    // LetS
f(){}()      // ExprS
if(true){let a = 5} //ExprS
```

(b) Statements that don't produce values

```
1 + 2        // InfixE
if(true){5}  // IfE
fn(x){x}(true) // CallE
```

(c) Expressions that produce values

```
if(false){} else {} // IfE
if(true){let a = 5} // IfE
f(){}()      // CallE
```

(d) Expressions that don't produce values

Figure 1: Statements, expressions, values

In addition, the same statement / expression can produce a value or not, depending on the bindings in the environment. A simple example for that is the statement / expression

```
if (x>0){let a=5}
```

If `x` is bound to a value for which `x>0` is "truthy", the if expression / expression statement is evaluated to `nil`, whereas it is evaluated to the Monkey object `NULL` otherwise.

This example also makes it pretty obvious that the claim that expressions have meanings, while statements don't makes little sense for the Monkey programming language, since expression statements get their meaning for expressions. Moreover, expressions can get their meaning from statements: the meaning of if expressions and function calls is typically derived from block statements.

[1] Interpreter Book: 33

1.2 Monkey's Object System

In the subset of the Monkey programming language implemented by the end of chapter 3, Monkey has the following types of objects:

Monkey-Type	Go-Type
`INTEGER`	`object.Integer`
`BOOLEAN`	`object.Boolean`
`RETURN_VALUE`	`object.ReturnValue`
`FUNCTION`	`object.Function`
`NULL`	`object.Null`
`ERROR`	`object.Error`

Figure 2: Object types in Monkey by end of chapter 3

1.3 `nil` and `NULL` and `Error` objects

When describing the foundation of Monkey's object system, the author promises that

> we're going to represent every value we encounter when evaluating Monkey source code as an `Object`, an interface of our design. Every value will be wrapped inside a struct, which fulfills this Object interface.[2]

1.3.1 `nil`

He doesn't live up to that promise:

```
func Eval(node ast.Node, env *object.Environment) object.Object {
    switch node := node.(type) {
    ...
    }

    return nil
}
```

Figure 3: Closing return statement of Eval-function

In the closing return statement of the central evaluation function, `nil` is returned if not specified differently in one of the cases of the preceding switch-statement. The only case, where this closing return statement is used is with regard to let functions. In addition, the special functions implementing the evaluation of programs and block statements return a nil value if they are empty (see code in Figure 4). In both cases, a nil interface is returned, if the statement list is empty.

[2] Interpreter Book: 108

```
func evalProgram(program *ast.Program, env *object.Environment)
object.Object {
    var result object.Object

    for _, statement := range program.Statements {
        result = Eval(statement, env)

        switch result := result.(type) {
        case *object.ReturnValue:
            return result.Value
        case *object.Error:
            return result
        }
    }
    return result
}
```

(a) evaluating programs

```
func evalBlockStatement(
    block *ast.BlockStatement,
    env *object.Environment,
) object.Object {
    var result object.Object

    for _, statement := range block.Statements {
        result = Eval(statement, env)

        if result != nil {
            rt := result.Type()
            if rt == object.RETURN_VALUE_OBJ
                                || rt == object.ERROR_OBJ {
                return result
            }
        }
    }
    return result
}
```

(b) evaluating programs

Figure 4: Evaluating programs and block statements

The author seems to be (somewhat) aware that nodes of the ast can evaluate to `nil`, since at some places, he checks whether a value is `nil` before using the `Type()`-function that each `Object` implements (at others, he doesn't, which leads to runtime errors for certain inputs) - not only in the evaluator-code, but also in the repl-code (Figure 5).

```
...
        evaluated := evaluator.Eval(program, env)
        if evaluated != nil {
            io.WriteString(out, evaluated.Inspect())
            io.WriteString(out, "\n")
        }
...
```

Figure 5: Taking care of `nil`-values in the repl

1.3.2 `NULL`

Relatively early, the author introduces `NULL` values. Null values "represent the absence of a value", he says and warns that "the language would be safer to use if it doesn't allow null or null references" and lets the reader know that the use of null values led to many "crashes".[3]

Unfortunately, with the introduction of `NULL`, the possibility of nodes evaluating to `nil` has not been abandoned. Moreover, I don't see that the evaluator ever crashes for using `NULL`, whereas it does so for using `nil`.

Although he still considers the possibility of nodes evaluating to `nil` - as becomes apparent by his questioning whether a value is `nil` in the code, he seems to not intend it. For if expressions, he explicitly states:

> When a conditional doesn't evaluate to a value it's supposed to return `NULL`, e.g.: `if (false) { 10 }` [4]

In the case of a missing alternative, this succeeds, but not in cases in which the condition is "truthy" and the consequence is a block statement evaluating to `nil` or in cases in which the condition is not truthy and and the alternative is a block statement evaluating to `nil`. Figure 6 shows some (pretty minimal) examples.

After the introduction of `NULL` values, we have both options: nodes can evaluate to `nil` as well as to `NULL`.

1.3.3 `Error` objects

In the following, error objects are introduced as a cure for the dilemma so far solved by returning `NULL`s.

The author intruduces a function `newError` that returns an error object (Figure 7) and comments:

[3] Interpreter Book: 108
[4] Interpreter Book: 125

```
if (true) {}
if (true) {} else {...}
if (false) {...} else {}

if (true) { let a = 5 }
let a = fn(){}(); if (true) {a}
```

Figure 6: If expressions evaluating to `nil`

```
func newError(format string, a ...interface{}) *object.Error {
    return &object.Error{Message: fmt.Sprintf(format, a...)}
}
```

Figure 7: error

> This newError function finds its use in every place where we didn't know what to do before and returned NULL instead[5]

Yet, he does not give up `NULL`-objects. An if expression with a non-truthy condition and non-existent alternative still evaluates to `NULL` and can thus pass `NULL`-values to many other expressions.

1.4 Conclusion

There are three co-existing ways in the implementation of the Monkey programming language to deal with situations where it is not immediately clear what a node is to evaluate to:

1. return `nil`
2. return `NULL`
3. return an `Error` object

That's maybe a bit (too) much.

`NULL` values are given birth by if expressions and `nil` values are given birth by programs, block statements and let statements. However, they can spread. Figure 8 gives an overview over wich nodes can evaluate to which of our unorthodox values for the subset of Monkey implemented by the end of chapter 3.

[5]Interpreter Book: 133

	nil	NULL	Error
Program	✓	✓	✓
LetStatement	✓	-	✓
ReturnStatement	-	-	✓
ExpressionStatement	✓	✓	✓
BlockStatement	✓	✓	✓
FunctionLiteral	-	-	-
Boolean	-	-	-
IntegerLiteral	-	-	-
PrefixExpression	-	-	✓
InfixExpression	-	-	✓
IfExpression	✓	✓	✓
CallExpression	✓	✓	✓
Identifier	✓	✓	✓

Figure 8: Unorthodox values

List of Figures