On ...

Stefanie

April 9, 2021

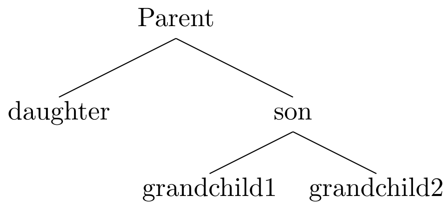
Abstract

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

List of Figures

List of Tables



$$\forall x(x \in \mathbb{N} \to x \neq \{\emptyset, 0\})$$

$$s(x) := \sum_{i=0}^{y} i$$

1 On Quality Assurance and Testing

- my first 2 tests are just minor stuff, but the nil-thing is important, since it pertains to the basics of the Monkey object system
  - proposal: test 2 possibilities
    - just get rid of nil and use NULL instead
    - go back to the idea of statements having no value, while expressions have
      typical repls: treat expressions differently from statements (ghci, for example)
- falsche Sicherheit
  - expl start of compiler book

2 On Evaluating to Unorthodox Values

2.1 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:

```
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.

Moreover, statements can get their values from expressions - take expression statements for example and expressions can get their values from statements - if expressions and function calls get it from block statements.

[1]Interpreter Book: 33

2.2 Monkey's Object System

After 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

2.3 `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]

2.3.1 `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 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 (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

2.3.2 `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`, it does so for using `nil`.

Although he still considers the possibility of nodes evaluating to `nil` - as becomes apparent in questioning whether a value is `nil`, 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 suceeds, but not, if the condition is "truthy" and its condition is a block statement evaluating to `nil` or if its condition is not truthy and and its alternative is a block statement evaluating to `nil`. Figure 6 shows some (pretty minimal) examples.

```
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`

So, now we have both options: nodes can evaluate to `nil` as well as to `NULL`.

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

2.3.3 `Error` objects

Error objects are introduced as a cure for the dilemma so far solved by returning `NULL`s.

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

Figure 7: error

The author intruduces a function `newError` that returns an error object (Figure 9) and comments:

> 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.

2.4 Conclusion

In Monkey, there are three ways to deal with situations where it is not so 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.

[5]Interpreter Book: 133

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

Figure 8: Unorthodox values

3 On Booleans [TODO]

4 On Environments and Closures [TODO]

5 On Function Calls

```
<call-expression>   ::= <identifier> LBRACE
                        <comma-separated-expressions> RBRACE
                    |   <function-literal> LBRACE
                        <comma-separated-expressions> RBRACE
```

(a) Rule described

```
<call-expression>   ::= <expression> LBRACE
                        [<comma-separated-expressions>] RBRACEs
```

(b) Rule implemented

Figure 9: error

- he describes the grammar rule differently from what he implements
- restricting the grammar would disable some nice usages
- where the function expression is restricted to expressions that are identifiers or function literals:
- *Function* can be can be any ‘Expression‘; there are no restrictions;
- only parsed via ‘parseExpression‘–¿ ‘parseCallExpression‘

  (whenever we have an expression not followed by an operator, but a left brace)

```
‘‘‘go
type CallExpression struct {
Token     token.Token // The ’(’ token
Function  Expression  // Identifier or FunctionLiteral
Arguments []Expression
}
‘‘‘
```

There are several places where the author seems to make clear that he only wants to allow *identifiers* and *function literals* in the function constituent of function calls. However, the way the parser is implemented any expression is allowed as this constituent.

5.1 Advantages

Moreover, one could come up with perfectly reasonable Monkey code where it makes sense to have expressions besides *identifiers* and *function literals* as function constituents of function calls:

```
\begin{lstlisting}
let func1 = fn...
let func2 = fn...
if(...){func1} else {func2}(...)
\end{lstlisting}
```

```
\begin{lstlisting}
let func1 = fn...
let func2 = fn...
let choose_func = fn(x){if (x==1){return func1} if (x==2){return func2}}
choose_func(1)(...)
\end{lstlisting}
```

Therefore, it is more reasonable to fix the evaluation of ast nodes.

5.2 Runtime Errors

And we have an obvious problem with function calls:

```
\begin{lstlisting}
if (true){}()

\end{lstlisting}
```

causes a runtime error.

So, one might want to correct the parser and implement it in such a way that it restricts the function constituent of function calls to *identifiers* and *function literals*.