THORSTEN BALL

THE LOST CHAPTER

A MACRO SYSTEM FOR MONKEY

THE LOST CHAPTER: A MACRO SYSTEM FOR MONKEY

THORSTEN BALL

CONTENTS

5.1 - Macro Systems

Macro systems are the features of programming languages that concern themselves with macros: how to define them, how to access them, how to evaluate them and how the macros themselves work. They can be divided into two broad categories: text-substitution macro systems and syntactic macro systems. In my mind, they are the search-and-replace and the code-as-data camps.

The first category, text-substitution macros, are arguably the simpler form. One example for this type of macro system is the C preprocessor. It allows you to generate and modify C code by using a separate macro language in the rest of your normal C code. It works by parsing and evaluating this separate language before the resulting code is then compiled by the C compiler. Here is a simple example:

```
#define GREETING "Hello there"

int main(int argc, char *argv[])
{
#ifdef DEBUG
  printf(GREETING " Debug-Mode!\n");
#else
  printf(GREETING " Production-Mode!\n");
#endif

  return 0;
}
```

The instructions for the preprocessor are the lines prefixed with `#`. In the first line we define a variable, `GREETING`, which will be replaced with `"Hello there"` in the rest of the source code. Quite literally, too, so you really have to pay attention to escaping and scoping concerns. In the third line we check whether the preprocessor variable `DEBUG` is defined - either by us, our build system, our compiler or the C libraries shipping with our operating system. Based on that either the `Debug-Mode` or the `Production-Mode` statements are produced.

It's a simple system that works remarkably well when used with care and restraint. But there are limits to what it can do, because its influence on the code it produces exists merely on a textual level. In that regard it's much more closer to a templating system than the macro systems of the second category, the syntactic macros.

These macro systems do not work with code as text, but they treat *code as data*. Sounds weird? Yes. If you're unaccustomed to it, this can be a pretty strange thought. But it's not hard to understand, I promise. It just needs a slight shift in perspective to be fully grasped.

In fact, we've already touched upon this in chapter 2 when we looked at how lexers and parsers turn source code from text into ASTs. ASTs represent source code with data structures other than strings. These data structures are available in the language in which the parser operates. In our case, we turned Monkey source code, initially a string, into the Go structs that make up our Monkey AST. And then we could treat the code as data: we could pass around, modify and generate Monkey source code inside our Go program.

In languages with syntactic macros you can do that *in the language itself*, not just in an outer host language. If a language X has a syntactic macro system, you can use language X to work with source code written in X. Just like we worked with Monkey source code while using Go. "Pass this if-expression to this function, take this function call and save it here, change the name used in this let-statement." The language becomes *self-aware*, so to speak, and with macros allows you to inspect and modify itself. Like a surgeon operating on themself. Nice, right?

This type of macro system was pioneered by Lisp and can now be found in a lot of its descendants: Common Lisp, Clojure, Scheme, Racket. But also non-Lisp languages like Elixir and Julia have elegant macro systems that are built on this idea of treating code as data and allowing macros to access it.

This is all still pretty abstract, so let's try to clear up some confusion by playing around with such a syntactic macro system. We're going to use Elixir, because its syntax is easy to read and understand. But the ideas and mechanisms apply to all of the languages mentioned above.

The first we need to understand is Elixir's `quote` function. It allows us to stop code from being evaluated - effectively turning code into data:

```
iex(1)> quote do: 10 + 5
{:+, [context: Elixir, import: Kernel], [10, 5]}
```

Here we pass the infix expression `10 + 5` to `quote` as a single argument in a `do` block. But instead of `10 + 5` being evaluated - as arguments in function calls normally are - `quote` returns a data structure that represents this very expression. It's a tuple containing the operator `:+`, meta information like the context of the call, and the list of operands `[10, 5]`. This is Elixir's AST and how Elixir represents code all throughout the language.

We can access it just like any other tuple:

```
iex(2)> exp = quote do: 10 + 5
{:+, [context: Elixir, import: Kernel], [10, 5]}
iex(3)> elem(exp, 0)
:+
iex(4)> elem(exp, 2)
[10, 5]
```

So `quote` allows us to stop the evaluation of code and treat code as data. That's already super interesting, but we can take it up a notch.

Let's say we want to use `quote` to build an AST node that represents an infix expression involving three integer literals. One of the numbers should be injected into the AST dynamically. It's bound to a name, `my_number`, and we only want to refer to it by this name. Here's a first attempt using `quote` that doesn't work:

```
iex(6)> my_number = 99
99
iex(7)> quote do: 10 + 5 + my_number
{:+, [context: Elixir, import: Kernel],
 [{:+, [context: Elixir, import: Kernel], [10, 5]}, {:my_number, [], Elixir}]}
```

And of course it doesn't. `quote` stops its argument from being evaluated. So `my_number` is just an identifier when passed to `quote`. It doesn't resolve to `99`, because it's not evaluated. For that, we need another function, called `unquote`:

```
iex(8)> quote do: 10 + 5 + unquote(my_number)
{:+, [context: Elixir, import: Kernel],
 [{:+, [context: Elixir, import: Kernel], [10, 5]}, 99]}
```

`unquote` allows us to "jump out of" the `quote` context and evaluate code. Here it causes the identifier `my_number` to evaluate to `99`.

These two, `quote` and `unquote`, are the tools Elixir gives us to influence when and how code is evaluated or left untouched and turned into data. Most often they are used inside macros, which we can define in Elixir with the keyword `defmacro`. Here is a simple example, a macro that turns infix expressions using a `+` operator into infix expressions using `-`, called `plus_to_minus`:

```
defmodule MacroExample do
  defmacro plus_to_minus(expression) do
    args = elem(expression, 2)

    quote do
      unquote(Enum.at(args, 0)) - unquote(Enum.at(args, 1))
    end
  end
end
```

The most important thing about macros in Elixir (and a lot of languages with this type of macro system) is this: everything that's passed to a macro as an argument is `quoted`. A macro's arguments are not evaluated and can be accessed like any other piece of data.

We do just that in the first line of `plus_to_minus`. We bind the arguments of the passed-in expression to `args` and then we use `quote` and `unquote` to construct the AST of an infix expression. Note: this new expression uses `-` to subtract the second argument from the first.

If this macro is called with a `10 + 5` as the argument, what comes out is not `15`, but this, the result of evaluating `10 - 5`:

```
iex(1)> MacroExample.plus_to_minus 10 + 5
5
```

Yes, we just modified code like it was data! That's much more powerful than the C preprocessor, isn't it? Here in the code-as-data camp is where things get interesting! Code as data? Code modifying itself? Surgeons operating on themselves? Macros writing code? Writing code that writes code? Sweet Monkey yes, I'm in!

Naturally, I decided that if Monkey should have a macro system, it needs to be of this kind. And this is what we're going to build. A syntactic macro system for Monkey that allows us to access, modify and generate Monkey source code.

Let's do this!

5.2 - A Macro System for Monkey

Adding macros to a programming language means, first and foremost, answering a lot of questions: "How exactly? Which consequences does this change have? What is this influenced by?" Having a clear picture of the outcome in mind keeps us from getting lost in these questions. So before we begin, as always, let's get a clear picture of what we actually want to build.

The macro system we're going to add to Monkey will be modelled after Elixir's, which itself is modelled after a simple `define-macro` system known from the Lisp and Scheme world.

The first things we're going to add are the `quote` and `unquote` functions. They will allow us to influence when exactly Monkey code is evaluated.

Here is what using `quote` in Monkey will look like:

```
$ go run main.go
Hello mrnugget! This is the Monkey programming language!
Feel free to type in commands
>> quote(foobar);
QUOTE(foobar)
>> quote(10 + 5);
QUOTE((10 + 5))
```

```
>> quote(foobar + 10 + 5 + barfoo);
QUOTE((((foobar + 10) + 5) + barfoo))
```

As you can see, `quote` will take one argument and stop it from being evaluated. It will return an object that represents the quoted code.

The matching `unquote` function will allow us to circumvent `quote`:

```
>> quote(8 + unquote(4 + 4));
QUOTE((8 + 8))
```

`unquote` will only be usable inside the expression that's passed to `quote`. But in there it will also be possible to `unquote` source code that's been `quote`d before:

```
>> let quotedInfixExpression = quote(4 + 4);
>> quotedInfixExpression;
QUOTE((4 + 4))
>> quote(unquote(4 + 4) + unquote(quotedInfixExpression));
QUOTE((8 + (4 + 4)))
```

We're going to need that when we put in the final piece of the macro system: the `macro` literals. They allow us to define macros:

```
>> let reverse = macro(a, b) { quote(unquote(b) - unquote(a)); };
>> reverse(2 + 2, 10 - 5);
1
```

Macro literals look just like function literals, except that the keyword is not `fn` but `macro`. And once a macro is bound to a name we can call it like a function, too. Except that these calls will be evaluated in a different way. Just like in Elixir the arguments won't be evaluated before being passed to the macro's body. Combined with the aforementioned ability to `unquote` code that's been `quote`d before, that allows us to selectively evaluate macro arguments, which are just `quote`d code:

```
>> let evalSecondArg = macro(a, b) { quote(unquote(b)) };
>> evalSecondArg(puts("not printed"), puts("printed"));
printed
```

By returning code that only contains the second argument, the `puts("printed")` expression, we effectively stop the first argument from being evaluated.

If any of these examples don't make sense yet, don't worry! That'll change. We'll see exactly how and why they work, because we're going to build the features they use ourselves, from scratch.

Of course, while building our macro system, we will have to make trade-offs. The biggest being that it won't be as polished and feature-complete as its production-ready counterparts in other languages. But we'll build a fully working macro system nonetheless. It'll be easy to understand and easy to extend, so we can always tweak, optimize and improve it in any way we want later on.

Let's write code that lets us write code that writes code!

5.3 - QUOTE

The first thing we are going to add is the `quote` function. `quote` will only be used inside macros and its purpose is simply stated: when called, it stops its argument from being evaluated. Instead it returns the AST node representing the argument.

How do we implement that? Let's start with the return value. Every function in Monkey returns values of the interface type `object.Object`. And `quote` can't be an exception here, since that would break our `Eval` function, which relies on every Monkey value being an `object.Object` and itself returns an `object.Object`.

So in order for `quote` to return an `ast.Node` we need a simple wrapper that allows us to pass around an `object.Object` containing an `ast.Node`. Here it is:

```go
// object/object.go

const (
// [...]

    QUOTE_OBJ = "QUOTE"
)

type Quote struct {
    Node ast.Node
}

func (q *Quote) Type() ObjectType { return QUOTE_OBJ }
func (q *Quote) Inspect() string {
    return "QUOTE(" + q.Node.String() + ")"
}
```

There's not much to it, right? `object.Quote` is just a thin wrapper around an `ast.Node`. But it allows us to take the next step: when we evaluate a call to `quote`, we now need to stop the argument of the call from being evaluated. Instead we need to wrap it in an `object.Quote` and return that instead. And that shouldn't pose a problem, since we have full control over what gets evaluated in our `Eval` function.

Let's write a simple test case that makes sure that exactly this happens when `quote` is called:

```go
// evaluator/quote_unquote_test.go

package evaluator

import (
    "testing"

    "monkey/object"
)

func TestQuote(t *testing.T) {
    tests := []struct {
        input    string
        expected string
    }{
        {
            `quote(5)`,
            `5`,
        },
    }

    for _, tt := range tests {
        evaluated := testEval(tt.input)
        quote, ok := evaluated.(*object.Quote)
        if !ok {
            t.Fatalf("expected *object.Quote. got=%T (%+v)",
```

```
                evaluated, evaluated)
        }

        if quote.Node == nil {
            t.Fatalf("quote.Node is nil")
        }

        if quote.Node.String() != tt.expected {
            t.Errorf("not equal. got=%q, want=%q",
                quote.Node.String(), tt.expected)
        }
    }
}
```

At first glance this looks just like any other test in the `evaluator` package, where we pass source code to `Eval` and expect it to return a certain type of object. And this test does that too. It passes the `tt.input` to `testEval` and expects it to return an `*object.Quote`. The difference comes at the end.

With the last assertion we make sure that the correct `ast.Node` is wrapped inside of that `*object.Quote` by comparing the return value of the node's `String()` method with the `tt.expected` string. That makes the tests really expressive and readable, because we don't have to build `ast.Nodes` by hand with verbose struct literals. The downside is that we're testing through another abstraction layer. That's okay in this case though, because we're confident in the simple `String()` methods of our `ast.Nodes`. We should keep their limitations in mind though.

Now that we know how this test function works, here are a few more test cases that make clearer how evaluating a call to `quote` doesn't evaluate its argument:

```
// evaluator/quote_unquote_test.go

func TestQuote(t *testing.T) {
    tests := []struct {
        input    string
        expected string
    }{
// [...]
        {
            `quote(5 + 8)`,
            `(5 + 8)`,
        },
        {
            `quote(foobar)`,
            `foobar`,
        },
        {
            `quote(foobar + barfoo)`,
            `(foobar + barfoo)`,
        },
    }
// [...]
}
```

Since the only thing we've implemented so far is the `object.Quote` definition, the tests fail:

```
$ go test ./evaluator
--- FAIL: TestQuote (0.00s)
  quote_unquote_test.go:37: expected *object.Quote. got=*object.Error\
     (&{Message:identifier not found: quote})
```

```
FAIL
FAIL    monkey/evaluator       0.009s
```

Here's what's happening to make this test fail. The parser first turns `quote()` calls into `*ast.CallExpressions`. `Eval` then takes these expressions and evaluates them just like any other `*ast.CallExpression`. That means, first of all, getting to the function that's being called. If the `Function` field of an `*ast.CallExpression` contains an `*ast.Identifier`, then `Eval` tries to look up the identifier in the current environment. In our case here, looking up `quote` doesn't yield a result and we get the `identifier not found: quote` error message.

A first approach would be to define a built-in function called `quote`. `Eval` would then find the function in the environment and try to call it. That's good, but the problem lies in `Eval`s default behaviour when calling functions. Remember what it does before it evaluates a function's body? *It evaluates the arguments of the call!* That's *exactly* what we don't want! `quote` is supposed to return its argument *unevaluated.*

What we need to do instead is to change this existing part of `Eval` so it doesn't evaluate the argument in `quote` call expressions:

```go
// evaluator/evaluator.go

func Eval(node ast.Node, env *object.Environment) object.Object {
  // [...]
    case *ast.CallExpression:
        function := Eval(node.Function, env)
        if isError(function) {
            return function
        }

        args := evalExpressions(node.Arguments, env)
        if len(args) == 1 && isError(args[0]) {
            return args[0]
        }

        return applyFunction(function, args)
  // [...]
}
```

The `evalExpressions(node.Arguments, env)` expression is what we need to skip in case we're calling `quote`. Let's do that, let's short-circuit `Eval`:

```go
// evaluator/evaluator.go

func Eval(node ast.Node, env *object.Environment) object.Object {
  // [...]
    case *ast.CallExpression:
        if node.Function.TokenLiteral() == "quote" {
            return quote(node.Arguments[0])
        }

  // [...]
}
```

We simply check whether we have a call to `quote` at hand by checking the `TokenLiteral()` method of the call expressions `Function` field. Granted, that's not the most beautiful solution, but it's all that's needed and for now does the job.

In case the call expression is indeed a `quote` call we pass the single argument to `quote` (remember, we said we'll only allow one argument to `quote`!) to a function that's also called `quote`. It looks like this:

```go
// evaluator/quote_unquote.go

func quote(node ast.Node) object.Object {
    return &object.Quote{Node: node}
}
```

I hope you didn't expect more. We simply take the argument and wrap it in a newly allocated `*object.Quote` and return that. And, would you look at that, it makes our tests pass!

```
$ go test ./evaluator
ok      monkey/evaluator        0.009s
```

Alright! `quote` works as expected. Great! Now we can start with the real fun stuff. Because `quote` is only the half of it and we need to build its partner in macro crime now: `unquote`.

5.4 - UNQUOTE

You know what they say: there can be no light without the dark, no Vim without Emacs and no `quote` without `unquote`. Or something like that.

If the idea behind `quote` is that its arguments are not evaluated and just stay `ast.Node`s, then `unquote` exists to punch holes in that idea. With `quote` we're telling `Eval`: *"skip this part"*. But with `unquote` we're adding *"…except this one here, evaluate this"*.

`unquote` allows us to evaluate expressions inside a call to `quote`. In practical terms: when we call `quote(8 + unquote(4 + 4))` we don't want a `ast.Node` representing `8 + unquote(4 + 4)` to be returned. Instead we want `8 + 8`, because `unquote` should evaluate its argument.

Thankfully, it's pretty easy to turn this desired behaviour into a test case:

```go
// evaluator/quote_unquote_test.go

func TestQuoteUnquote(t *testing.T) {
    tests := []struct {
        input    string
        expected string
    }{
        {
            `quote(unquote(4))`,
            `4`,
        },
        {
            `quote(unquote(4 + 4))`,
            `8`,
        },
        {
            `quote(8 + unquote(4 + 4))`,
            `(8 + 8)`,
        },
        {
            `quote(unquote(4 + 4) + 8)`,
            `(8 + 8)`,
        },
    }

    for _, tt := range tests {
```

```go
        evaluated := testEval(tt.input)
        quote, ok := evaluated.(*object.Quote)
        if !ok {
            t.Fatalf("expected *object.Quote. got=%T (%+v)",
                evaluated, evaluated)
        }

        if quote.Node == nil {
            t.Fatalf("quote.Node is nil")
        }

        if quote.Node.String() != tt.expected {
            t.Errorf("not equal. got=%q, want=%q",
                quote.Node.String(), tt.expected)
        }
    }
}
```

The mechanism here is the same as in the `TestQuote` function we wrote earlier. We pass `tt.input` to `testEval` and then compare the output of the quoted `ast.Nodes` `String()` method against our `tt.expected` value. The difference is that we now we have `unquote` calls inside the calls to `quote`. That makes the tests fail, as expected:

```
$ go test ./evaluator
--- FAIL: TestQuoteUnquote (0.00s)
  quote_unquote_test.go:88: not equal. got="unquote(4)", want="4"
  quote_unquote_test.go:88: not equal. got="unquote((4 + 4))", want="8"
  quote_unquote_test.go:88: not equal. got="(8 + unquote((4 + 4)))",\
    want="(8 + 8)"
  quote_unquote_test.go:88: not equal. got="(unquote((4 + 4)) + 8)",\
    want="(8 + 8)"
FAIL
FAIL    monkey/evaluator        0.009s
```

Making them pass might seems easy. We already know how to evaluate things! How hard can it be to evaluate calls to `unquote`? We already have a `case` branch for `*ast.CallExpression` in place in `Eval`, we can add another conditional, just like we did for `quote`.

But exactly therein lies the rub.

We can't just tweak `Eval` again. Because we never call `Eval`! Remember: when we come across a `quote` call, its argument is wrapped inside an `*object.Quote` and, as desired, *not passed to `Eval`*. And since `unquote` calls are only allowed inside the argument of a `quote` call, `Eval` will never come across them. We can't rely on the recursive nature of `Eval` to find `unquote` calls for us and evaluate them. We have to do it by hand.

In other words: making the tests pass requires us to traverse the argument passed to `quote`, find the calls to `unquote` and `Eval` their arguments. The good news is that it's not hard to do. We already did it - in `Eval` - and now we need to do it again. With one small change: we need to modify nodes as we walk along the AST.

Walking the tree

The word "modify" requires some explanation. We start by traversing the AST, find calls to `unquote` and pass the argument of the call to `Eval`. Nothing is modified so far. Only after the argument is evaluated comes the "modify" part: we now want to replace the whole `*ast.CallExpression` involving `unquote` with the result of this call to `Eval`.

The problem is that this means replacing an `*ast.CallExpression` with the return value of `Eval`, an `object.Object`. Our compiler won't allow that. The solution is to turn the result of the `unquote` call into a new AST node and replace (modify!) the existing call to `unquote` with this newly created AST node.

Trust me, it'll make sense soon.

In order for us to do all this by hand, without the help of `Eval`, and to make `unquote` work, we're going to build a generic function that allows us to do AST traversal with the possible modification and replacement of `ast.Node`s. It's generic and not `unquote`-specific, because we'll need it again later on, once we have `quote` and `unquote` in place and need to take care of macros. It also makes the code much nicer.

Now, is there a better place to add such a function than our old friend, the `ast` package itself?

FIRST STEPS

Here is what we want the function to do:

```
// ast/modify_test.go

package ast

import (
    "reflect"
    "testing"
)

func TestModify(t *testing.T) {
    one := func() Expression { return &IntegerLiteral{Value: 1} }
    two := func() Expression { return &IntegerLiteral{Value: 2} }

    turnOneIntoTwo := func(node Node) Node {
        integer, ok := node.(*IntegerLiteral)
        if !ok {
            return node
        }

        if integer.Value != 1 {
            return node
        }

        integer.Value = 2
        return integer
    }

    tests := []struct {
        input    Node
        expected Node
    }{
        {
            one(),
            two(),
        },
        {
            &Program{
                Statements: []Statement{
```

```go
                    &ExpressionStatement{Expression: one()},
                },
            },
            &Program{
                Statements: []Statement{
                    &ExpressionStatement{Expression: two()},
                },
            },
        },
    }

    for _, tt := range tests {
        modified := Modify(tt.input, turnOneIntoTwo)

        equal := reflect.DeepEqual(modified, tt.expected)
        if !equal {
            t.Errorf("not equal. got=%#v, want=%#v",
                modified, tt.expected)
        }
    }
}
```

That's quite a test setup, so let's take a closer look at what we want to happen here.

Before we define our `tests` we define two helper functions: `one` and `two`. Both return fresh `*ast.IntegerLiterals`, which wrap the numbers `1` and `2` respectively. `one` and `two` exist so we don't have to construct integer literals again and again in the test cases themselves. That makes our tests slightly more readable.

Next we define a function called `turnOneIntoTwo`. This one has an interesting interface: it accepts an `ast.Node` and returns an `ast.Node`. And it checks whether the passed-in `ast.Node` is an `*ast.IntegerLiteral` representing a `1`. If that's the case, it turns the `1` into a `2`. In other words: it "modifies" an `ast.Node`. It's easy to write and to understand. There's not much that can go wrong with it. That's why it's our simple test helper we pass to the yet-to-be-written-by-us function `ast.Modify` for each test case.

In the first test case, the input consists solely of the node returned by `one`. We expect that passing this node along with `turnOneIntoTwo` to `Modify` turns it into a `two`. That's pretty simple: a node comes in and if it matches certain criteria it's modified and returned.

In the second test case we expect more of `ast.Modify`: we want it to walk the given `ast.Program` tree and pass each child node to `turnOneIntoTwo`, which can then check if it's a `one` and turn it into a `two`.

I bet you can already see how this relates to our use case of finding calls to `unquote` and replacing them with a new AST node.

The tests fail, of course, because `ast.Modify` doesn't exist yet:

```
$ go test ./ast
# monkey/ast
ast/modify_test.go:49: undefined: Modify
FAIL    monkey/ast [build failed]
```

Thanks to the power of recursion (chant this three times for good luck!), making both test cases pass doesn't take a lot of code:

```go
// ast/modify.go
```

```go
type ModifierFunc func(Node) Node

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

    case *Program:
        for i, statement := range node.Statements {
            node.Statements[i], _ = Modify(statement, modifier).(Statement)
        }

    case *ExpressionStatement:
        node.Expression, _ = Modify(node.Expression, modifier).(Expression)
    }

    return modifier(node)
}
```

Yep, that's all it takes:

```
$ go test ./ast
ok      monkey/ast      0.007s
```

There are two ideas in `ast.Modify` that make it work.

The first one: recursively walk down the children of any given `ast.Node`. That's what happens in the `switch` statement and we already know this mechanism from our `Eval` function. But here certain `ast.Node`s do not and won't have their own `case` branch, e.g. `*ast.IntegerLiteral`. That's because we can't traverse their children, even if we wanted to, because they don't have any. But if they have children, as is the case with `*ast.Program`, we call `ast.Modify` with each child, which again could result in calls to `ast.Modify` with the children of the child, and so on. Recursion, huh?

An important effect of this recursive calling of `ast.Modify` is that we replace the node used as argument of the call with the node returned by the call. Which brings us to the second idea behind `ast.Modify`.

On the last line of `ast.Modify` it calls the `modifier` with the given `Node` and *returns* the result. That's important. If we'd only call `modifier(node)` and then `return node`, we wouldn't be able to replace nodes in the AST, but only mutate them.

The other effect of that last line is to stop the recursion. If we end up here, we don't have any more children we can traverse and return.

Completing the Walk

With that, the architecture of `ast.Modify` is in place. Now we just need to fill in the blanks and complete it, so it can traverse a complete `*ast.Program` that contains every type of `ast.Node` that we have.

Granted, what follows is not the most exciting part of our journey, but there are some subtleties to watch out for.

Infix Expressions

The test cases for modifying infix expressions look like this:

```go
// ast/modify_test.go

func TestModify(t *testing.T) {
// [...]

    tests := []struct {
        input    Node
        expected Node
    }{
// [...]
        {
            &InfixExpression{Left: one(), Operator: "+", Right: two()},
            &InfixExpression{Left: two(), Operator: "+", Right: two()},
        },
        {
            &InfixExpression{Left: two(), Operator: "+", Right: one()},
            &InfixExpression{Left: two(), Operator: "+", Right: two()},
        },
    }

// [...]
}
```

The main point here is to make sure that `ast.Modify` traverses and possibly modifies both arms, `Left` and `Right`, of an `*ast.InfixExpression`. As of now, it doesn't:

```
$ go test ./ast
--- FAIL: TestModify (0.00s)
        modify_test.go:62: not equal. got=&ast.InfixExpression{[...]},\
         want=&ast.InfixExpression{[...]}
        modify_test.go:62: not equal. got=&ast.InfixExpression{[...]},\
         want=&ast.InfixExpression{[...]}
FAIL
FAIL    monkey/ast      0.006s
```

I've removed some parts of the failing test output here and replaced them with `[...]` to not waste space. I'll refrain from even showing the failing test output in the remainder of this section.

The tests fail because the `one` integer literal hasn't been replaced with the `two`. Fixing that means adding a new `case` branch to `ast.Modify`:

```go
// ast/modify.go

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

// [...]
    case *InfixExpression:
        node.Left, _ = Modify(node.Left, modifier).(Expression)
        node.Right, _ = Modify(node.Right, modifier).(Expression)

    }

// [...]
}
```

That makes the tests pass and we can move on.

Prefix Expressions

This is the test case for prefix expressions:

```go
// ast/modify_test.go

func TestModify(t *testing.T) {
// [...]

    tests := []struct {
        input    Node
        expected Node
    }{
// [...]
        {
            &PrefixExpression{Operator: "-", Right: one()},
            &PrefixExpression{Operator: "-", Right: two()},
        },
    }

// [...]
}
```

And here is the `case` branch that makes them pass:

```go
// ast/modify.go

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

// [...]
    case *PrefixExpression:
        node.Right, _ = Modify(node.Right, modifier).(Expression)

    }

// [...]
}
```

Index Expressions

Index expressions also have two "arms", which we need to check in the tests:

```go
// ast/modify_test.go

func TestModify(t *testing.T) {
// [...]

    tests := []struct {
        input    Node
        expected Node
    }{
// [...]
        {
            &IndexExpression{Left: one(), Index: one()},
            &IndexExpression{Left: two(), Index: two()},
        },
    }
```

```go
// [...]
}
```

Walking the `Left` and `Index` nodes is easy enough:

```go
// ast/modify.go

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

// [...]
    case *IndexExpression:
        node.Left, _ = Modify(node.Left, modifier).(Expression)
        node.Index, _ = Modify(node.Index, modifier).(Expression)

    }

// [...]
}
```

If Expression

If-expression have quite a few moving parts that we need to traverse and possibly modify. They have the `Condition`, which can be any `ast.Expression`, and then they also have the `Consequence` and `Alternative` fields. Those are `*ast.BlockStatements`, which themselves can contain an arbitrary number of `ast.Statements`. The test case makes sure that all of these are traversed correctly:

```go
// ast/modify_test.go

func TestModify(t *testing.T) {
// [...]

    tests := []struct {
        input    Node
        expected Node
    }{
// [...]
        {
            &IfExpression{
                Condition: one(),
                Consequence: &BlockStatement{
                    Statements: []Statement{
                        &ExpressionStatement{Expression: one()},
                    },
                },
                Alternative: &BlockStatement{
                    Statements: []Statement{
                        &ExpressionStatement{Expression: one()},
                    },
                },
            },
            &IfExpression{
                Condition: two(),
                Consequence: &BlockStatement{
                    Statements: []Statement{
                        &ExpressionStatement{Expression: two()},
                    },
```

```go
                },
                Alternative: &BlockStatement{
                    Statements: []Statement{
                        &ExpressionStatement{Expression: two()},
                    },
                },
            },
        },
    }

// [...]
}
```

Thankfully, making this test case green takes a lot less lines:

```go
// ast/modify.go

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

// [...]
    case *IfExpression:
        node.Condition, _ = Modify(node.Condition, modifier).(Expression)
        node.Consequence, _ = Modify(node.Consequence, modifier).(*BlockStatement)
        if node.Alternative != nil {
            node.Alternative, _ = Modify(node.Alternative, modifier).(*BlockStatement)
        }

    case *BlockStatement:
        for i, _ := range node.Statements {
            node.Statements[i], _ = Modify(node.Statements[i], modifier).(Statement)
        }

    }

// [...]
}
```

Return Statement

Return statements have one child: the `ReturnValue`, which is an `ast.Expression`.

```go
// ast/modify_test.go

func TestModify(t *testing.T) {
// [...]

    tests := []struct {
        input    Node
        expected Node
    }{
// [...]
        {
            &ReturnStatement{ReturnValue: one()},
            &ReturnStatement{ReturnValue: two()},
        },
    }
```

```go
// [...]
}
```

That's a cute little test case, isn't it? Now take a look at this super cute `case` branch that makes it pass:

```go
// ast/modify.go

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

// [...]
    case *ReturnStatement:
        node.ReturnValue, _ = Modify(node.ReturnValue, modifier).(Expression)

    }

// [...]
}
```

I know, I know. This is not "super cute" and frankly this is getting boring. We're nearly done, though. I promise.

Let Statement

Let statements also only have one moving part: the `Value` they're binding to a name.

```go
// ast/modify_test.go

func TestModify(t *testing.T) {
// [...]

    tests := []struct {
        input    Node
        expected Node
    }{
// [...]
        {
            &LetStatement{Value: one()},
            &LetStatement{Value: two()},
        },
    }

// [...]
}
```

The `case` branch for `*ast.LetStatement` passes this `Value` to the `modifier` function:

```go
// ast/modify.go

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

// [...]
    case *LetStatement:
        node.Value, _ = Modify(node.Value, modifier).(Expression)

    }
```

```go
// [...]
}
```

Whew! We're done with statements! Let's take care of more literals!

Function Literal

Function literals have a `Body`, which is an `*ast.BlockStatement`, and `Parameters`, which are a slice of `*ast.Identifiers`. Traversing these parameters is optional, strictly speaking. The `ast.ModifierFunc` could do that itself, since it gets passed the function literal and the parameters can't contain any more children. But because we're nice we'll take care that, even though we can't easily test this here:

```go
// ast/modify_test.go

func TestModify(t *testing.T) {
// [...]

    tests := []struct {
        input    Node
        expected Node
    }{
// [...]
        {
            &FunctionLiteral{
                Parameters: []*Identifier{},
                Body: &BlockStatement{
                    Statements: []Statement{
                        &ExpressionStatement{Expression: one()},
                    },
                },
            },
            &FunctionLiteral{
                Parameters: []*Identifier{},
                Body: &BlockStatement{
                    Statements: []Statement{
                        &ExpressionStatement{Expression: two()},
                    },
                },
            },
        },
    }

// [...]
}
```

Since we already have a `case` branch for `*ast.BlockStatement`, it doesn't take a lot of lines to make this new test case pass:

```go
// ast/modify.go

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

// [...]
    case *FunctionLiteral:
        for i, _ := range node.Parameters {
            node.Parameters[i], _ = Modify(node.Parameters[i], modifier).(*Identifier)
```

```go
        }
        node.Body, _ = Modify(node.Body, modifier).(*BlockStatement)

    }

// [...]
}
```

Array Literal

Array literals are comma-separated lists of expressions. We only have to test that all of the expressions are iterated and passed to `ast.Modify` correctly:

```go
// ast/modify_test.go

func TestModify(t *testing.T) {
// [...]

    tests := []struct {
        input    Node
        expected Node
    }{
// [...]
        {
            &ArrayLiteral{Elements: []Expression{one(), one()}},
            &ArrayLiteral{Elements: []Expression{two(), two()}},
        },
    }

// [...]
}
```

A loop is all it takes to make this test case pass:

```go
// ast/modify.go

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

// [...]
    case *ArrayLiteral:
        for i, _ := range node.Elements {
            node.Elements[i], _ = Modify(node.Elements[i], modifier).(Expression)
        }

    }

// [...]
}
```

Hash Literal

Hash literals have one field we have to traverse, called `Pairs`, which is a `map[Expression]Expression`. That means we have to iterate over the map and modify both the keys and the values of the map, since both could contain a node we want to modify.

That itself is not a problem, but the test for this does not fit into our existing framework. Due to the way `reflect.DeepEqual` works with maps having pointers for keys and values, which I'll not get into here, we need a separate section for `*ast.HashLiterals` at the end of `TestModify` that doesn't use `reflect.DeepEqual`:

```go
// ast/modify_test.go

func TestModify(t *testing.T) {
// [...]

    hashLiteral := &HashLiteral{
        Pairs: map[Expression]Expression{
            one(): one(),
            one(): one(),
        },
    }

    Modify(hashLiteral, turnOneIntoTwo)

    for key, val := range hashLiteral.Pairs {
        key, _ := key.(*IntegerLiteral)
        if key.Value != 2 {
            t.Errorf("value is not %d, got=%d", 2, key.Value)
        }
        val, _ := val.(*IntegerLiteral)
        if val.Value != 2 {
            t.Errorf("value is not %d, got=%d", 2, val.Value)
        }
    }
}
```

Even though this is new, it's easy to understand for us. We create a new `*ast.HashLiteral` with only `ones` in its `Pairs`. This hash literal is then passed to `ast.Modify`, after which we assert, by hand, that every `one` has been effectively turned into a `two`. At the moment this doesn't work:

```
$ go test ./ast
--- FAIL: TestModify (0.00s)
  modify_test.go:146: value is not 2, got=1
  modify_test.go:150: value is not 2, got=1
  modify_test.go:146: value is not 2, got=1
  modify_test.go:150: value is not 2, got=1
FAIL
FAIL    monkey/ast      0.007s
```

The fix for this involves creating a new `map[Expression]Expression` we can replace `Pairs` with:

```go
// ast/modify.go

func Modify(node Node, modifier ModifierFunc) Node {
    switch node := node.(type) {

// [...]
    case *HashLiteral:
        newPairs := make(map[Expression]Expression)
        for key, val := range node.Pairs {
            newKey, _ := Modify(key, modifier).(Expression)
            newVal, _ := Modify(val, modifier).(Expression)
            newPairs[newKey] = newVal
        }
        node.Pairs = newPairs
```

```
    }

  // [...]
}
```

That makes the tests pass:

```
$ go test ./ast
ok      monkey/ast      0.006s
```

And with that our new `ast.Modify` function is done! Whew! We can now move on. But before we do, I need to tell you something.

Underscores are TODOs

Error handling! Let's make this quick: we straight up ignored it. Instead of making sure that our type assertions in `ast.Modify` work, we simply used the dreaded _ to ignore possible errors. Of course, that's not how it *should* be done and the reason why it's not done correctly is … space. I decided that it takes up too much space to show the complete error handling here, which would be full of rather boring conditionals and boolean checks.

So, before we go outside to dance and sing songs about our how our `ast.Modify` is finally working, please keep the _ of `ast.Modify` in the back of your mind.

That being said: yes! We did it! We successfully built `ast.Modify`!

Replacing Unquote Calls

With `ast.Modify` in place and fully tested, we can now turn our attention back to our original task. Remember? We need to evaluate `unquote` arguments in unevaluated, `quote`ed `ast.Node`s. If that doesn't jog your memory, maybe this still failing test does:

```
$ go test ./evaluator
--- FAIL: TestQuoteUnquote (0.00s)
  quote_unquote_test.go:88: not equal. got="unquote(4)", want="4"
  quote_unquote_test.go:88: not equal. got="unquote((4 + 4))", want="8"
  quote_unquote_test.go:88: not equal. got="(8 + unquote((4 + 4)))",\
    want="(8 + 8)"
  quote_unquote_test.go:88: not equal. got="(unquote((4 + 4)) + 8)",\
    want="(8 + 8)"
FAIL
FAIL    monkey/evaluator        0.007s
```

So, what do we have to do to make `TestQuoteUnquote` pass? Thinking in terms of `ast.Modify` this is fairly easy to articulate. Whenever we `quote` an `ast.Node` we need to pass it to `ast.Modify` first. The second argument to `ast.Modify`, an `ast.ModifierFunc`, then needs to replace calls to `unquote`.

Let's take a first step:

```go
// evaluator/quote_unquote.go

import (
    "monkey/ast"
    "monkey/object"
)
```

```go
func quote(node ast.Node) object.Object {
    node = evalUnquoteCalls(node)
    return &object.Quote{Node: node}
}

func evalUnquoteCalls(quoted ast.Node) ast.Node {
    return ast.Modify(quoted, func(node ast.Node) ast.Node {
        if !isUnquoteCall(node) {
            return node
        }

        call, ok := node.(*ast.CallExpression)
        if !ok {
            return node
        }

        if len(call.Arguments) != 1 {
            return node
        }

        return node
    })
}

func isUnquoteCall(node ast.Node) bool {
    callExpression, ok := node.(*ast.CallExpression)
    if !ok {
        return false
    }

    return callExpression.Function.TokenLiteral() == "unquote"
}
```

The change to our existing `quote` function is minimal. We simply pass the `node` to the new `evalUnquoteCalls` function before it's quoted.

`evalUnquoteCalls` then uses `ast.Modify` to traverse every `ast.Node` contained in the `quoted` parameter. And the `ast.ModifierFunc` checks if any given `ast.Node` is a call to `unquote` with one argument. That's right, for now the modifier function doesn't really do anything. It just checks which `node` it received; it doesn't modify anything! And, well, that's not enough to make the tests pass:

```
$ go test ./evaluator
--- FAIL: TestQuoteUnquote (0.00s)
  quote_unquote_test.go:88: not equal. got="unquote(4)", want="4"
  quote_unquote_test.go:88: not equal. got="unquote((4 + 4))", want="8"
  quote_unquote_test.go:88: not equal. got="(8 + unquote((4 + 4)))",\
    want="(8 + 8)"
  quote_unquote_test.go:88: not equal. got="(unquote((4 + 4)) + 8)",\
    want="(8 + 8)"
FAIL
FAIL    monkey/evaluator        0.007s
```

What do we need to do once we find an `unquote` call? `unquote` exists to punch holes into `quote`. That means that in contrast to `quote`, which stops its argument from being evaluated, it instead should evaluate it! And we already know how to do that – with a call to `Eval`!

But in order to use `Eval` we also need an `*object.Environment`, in which we can evaluate nodes. We have one at hand when `quote` is called, we just need to pass it through. For that we first we have to change our `case` branch in `Eval` and add the additional argument to the call to `quote`:

```
// evaluator/evaluator.go

func Eval(node ast.Node, env *object.Environment) object.Object {
// [...]

    case *ast.CallExpression:
        if node.Function.TokenLiteral() == "quote" {
            return quote(node.Arguments[0], env)
        }

// [...]
}
```

Now we can change the signature of `quote` and pass `env` on to `evalUnquoteCalls`:

```
// evaluator/quote_unquote.go

func quote(node ast.Node, env *object.Environment) object.Object {
    node = evalUnquoteCalls(node, env)
    return &object.Quote{Node: node}
}
```

And in the anonymous function in `evalUnquoteCalls` we can finally call `Eval` with the passed in `env`:

```
// evaluator/quote_unquote.go

func evalUnquoteCalls(quoted ast.Node, env *object.Environment) ast.Node {
    return ast.Modify(quoted, func(node ast.Node) ast.Node {
// [...]

        return Eval(call.Arguments[0], env)
    })
}
```

Perfect! Except that this doesn't work. The compiler rightfully refuses to accept our files:

```
$ go test ./evaluator
# monkey/evaluator
evaluator/quote_unquote.go:28: cannot use Eval(call.Arguments[0], env)\
  (type object.Object) as type ast.Node in return argument:
        object.Object does not implement ast.Node (missing String method)
FAIL    monkey/evaluator [build failed]
```

Just like we predicted it would earlier in this chapter. The newly inserted call to `Eval` returns an `object.Object`. And that doesn't work as the return value of our `ast.ModifierFunc`, which must return an `ast.Node`. We have an `object.Object` at hand but need an `ast.Node`.

Solving this is the last piece in the quote/unquote puzzle. Let's take a step back here and analyze what we need to do.

Our Go function `quote` returns an `*object.Quote`, containing an unevaluated `ast.Node`. Inside this unevaluated node the Monkey function `unquote` can be called to evaluate expressions. This works by evaluating the argument of the `unquote` call and replacing the whole call expression, an `ast.Node`, with the result of that evaluation. That result is an `object.Object`, which `Eval` returns.

That means, in order to replace the `unquote` call and to insert the result back into the unevaluated `ast.Node`, we have to convert it into an `ast.Node` again!

```go
// evaluator/quote_unquote.go

import (
// [...]
    "fmt"
    "monkey/token"
)

func evalUnquoteCalls(quoted ast.Node, env *object.Environment) ast.Node {
    return ast.Modify(quoted, func(node ast.Node) ast.Node {
// [...]

        unquoted := Eval(call.Arguments[0], env)
        return convertObjectToASTNode(unquoted)
    })
}

func convertObjectToASTNode(obj object.Object) ast.Node {
    switch obj := obj.(type) {
    case *object.Integer:
        t := token.Token{
            Type:    token.INT,
            Literal: fmt.Sprintf("%d", obj.Value),
        }
        return &ast.IntegerLiteral{Token: t, Value: obj.Value}

    default:
        return nil
    }
}
```

The new `convertObjectToASTNode` function creates `ast.Nodes` that represent the given `obj`. It also has to create a matching `token.Token`, or otherwise our tests would break (since the `String()` methods of our `ast.Nodes` heavily rely on the tokens). That's not the best of reasons and the constructions of the tokens is maybe best not done here, but it's one of the trade-offs we're making. Because besides the tokens, we're also ignoring possible errors and just return `nil`. But, you know, exercise for the reader and not dwelling on mistakes and all that…

Yes, that makes the tests pass!

```
$ go test ./evaluator
ok      monkey/evaluator        0.009s
```

`quote` and `unquote` work! We can now stop source code from being evaluated by using `quote` and we can make exceptions from that by evaluating certain nodes with `unquote`. Fantastic!

And that's not even all of i! There's a hidden feature. You may have noticed that in `evalUnquoteCalls` we have access to the current environment of the `quote` call, `env`, and then pass that to the `Eval` call in our `ast.ModifierFunc`. Yes, that allows us to do environment-aware evaluation inside `unquote` calls. Here are two test cases for `TestQuoteUnquote` that show what this makes possible:

```go
// evaluator/quote_unquote_test.go

func TestQuoteUnquote(t *testing.T) {
    tests := []struct {
        input    string
        expected string
    }{
```

```go
// [...]
        {
            `let foobar = 8;
            quote(foobar)`,
            `foobar`,
        },
        {
            `let foobar = 8;
            quote(unquote(foobar))`,
            `8`,
        },
    }

// [...]
}
```

In the first test we make sure that quoting an identifier doesn't resolve it, i.e. doesn't evaluate it. That's the sanity check.

But in the second test we use `unquote` to evaluate the identifier `foobar` with the `env` of the test passed to `Eval`. That in turn resolves the identifier and returns the object it's bound to. And that object then gets turned back into an AST node. Amazing, isn't it? The fact that we have the environment at our hands gives us quite a lot more power later on.

The only problem is that `convertObjectToASTNode` only knows how to convert integers back to AST nodes. Let's add some more tests and extend `convertObjectToASTNode` so that it can at least convert more than one type of object.

CONVERTING BOOLEANS TO AST NODES

Turning an `*oject.Boolean` back into an `ast.Node` is nearly as easy as turning integers into AST nodes. Here are two tests that make sure that we can handle the `true` literal and that we can also handle boolean results of expressions:

```go
// evaluator/quote_unquote_test.go

func TestQuoteUnquote(t *testing.T) {
    tests := []struct {
        input    string
        expected string
    }{
// [...]
        {
            `quote(unquote(true))`,
            `true`,
        },
        {
            `quote(unquote(true == false))`,
            `false`,
        },
    }

// [...]
}
```

The test fails because `convertObjectToASTNode` doesn't know how to handle booleans yet:

```
$ go test ./evaluator
--- FAIL: TestQuoteUnquote (0.00s)
  quote_unquote_test.go:101: quote.Node is nil
FAIL
FAIL    monkey/evaluator        0.009s
```

All we need to do is to add another `case` branch to the `switch` statement in `convertObjectToASTNode`:

```go
// evaluator/quote_unquote.go

func convertObjectToASTNode(obj object.Object) ast.Node {
    switch obj := obj.(type) {
// [...]

    case *object.Boolean:
        var t token.Token
        if obj.Value {
            t = token.Token{Type: token.TRUE, Literal: "true"}
        } else {
            t = token.Token{Type: token.FALSE, Literal: "false"}
        }
        return &ast.Boolean{Token: t, Value: obj.Value}
// [...]
    }
}
```

The tests pass. And I'm pretty sure that we now won't have any problems adding more types of objects to `convertObjectToASTNode`. There is one possible addition, though, that is so neat that I *have* to show it to you.

QUOTE INSIDE UNQUOTE INSIDE QUOTE

Here's the idea: we can add support for `quote` calls inside `unquote` inside `quote` solely by modifying `convertObjectToASTNode`. Cool, right? Alright, granted, "unquoting quoted source code inside quoted source code" is a great title for a book about meta-programming, but not a great explanation of what I mean.

Let me show you the tests for this. They provide some clarity:

```go
// evaluator/quote_unquote_test.go

func TestQuoteUnquote(t *testing.T) {
    tests := []struct {
        input    string
        expected string
    }{
// [...]
        {
            `quote(unquote(quote(4 + 4)))`,
            `(4 + 4)`,
        },
        {
            `let quotedInfixExpression = quote(4 + 4);
            quote(unquote(4 + 4) + unquote(quotedInfixExpression))`,
            `(8 + (4 + 4))`,
        },
    }
```

```
// [...]
}
```

In both test cases we first quote an infix expression, `4 + 4`, and then use it as an argument to call `unquote`, which itself is the argument of the outer `quote` call.

Especially the second test case makes clear what we want from this feature: passing quoted source code around. Adding support for previously `quote`d source code to `unquote` allows us to build up `ast.Node`s from multiple other `ast.Node`s. That'll come in handy soon enough, when we start building macros, which make use of this exact mechanism.

But first, we have to fix the failing tests, because `unquote` can't handle `*object.Quote` yet:

```
$ go test ./evaluator
--- FAIL: TestQuoteUnquote (0.00s)
  quote_unquote_test.go:110: quote.Node is nil
FAIL
FAIL    monkey/evaluator        0.007s
```

And here is what makes them pass and this addition so neat:

```
// evaluator/quote_unquote.go

func convertObjectToASTNode(obj object.Object) ast.Node {
    switch obj := obj.(type) {
// [...]

    case *object.Quote:
        return obj.Node

// [...]
    }
}
```

Two lines. It really is neat, isn't it? And if you don't understand how and why this `quote(unquote(quote()))` business works the way it does: worry not, that's normal, it takes a few glances to wrap ones head around.

Before we move on to the finale of our excursion into the world of meta-programming and macro systems - the macro expansion phase - I feel obliged to point out a few things our `quote/unquote` system is missing.

Words of Caution

It's simply outside the scope here, but what we're missing is the *proper* modification of AST nodes. At the moment `ast.Modify` simply modifies child nodes, but doesn't update the `Token` fields of the parent nodes. That leads to an inconsistent AST with nodes that may output the wrong information in their `String()` methods or even lead to bugs.

In `convertObjectToASTNode` we create new tokens on the fly. That's not a problem at the moment, but if our tokens would contain information about their origin, such as filename or line number, then we'd also have to update these here, which might be quite difficult for tokens that are created dynamically.

And, of course, the error handling, too, is neither "proper" nor "defensive", but rather "fingers crossed".

Alright, now that I've done my duty and warned you about the things lurking in the shadows, we're ready to walk through the final gate of macro systems and build our macro expansion phase.

5.5 - Macro Expansion

We interpret Monkey source code in a series of steps. We first give it to our lexer to turn it into tokens. Then comes the parser and turns the tokens into an AST. Finally, `Eval` takes this AST and evaluates its nodes recursively, statement by statement, expression by expression. That's three separate steps or phases. Lexing, parsing and evaluation. Speaking in data structures: strings to tokens, tokens to AST, AST to output.

What we're going to do next is add another phase. The macro expansion phase. It will sit right between the second and third one, between parsing and evaluation. And it couldn't sit anywhere else; there's a necessity to this position. The reason for that lies in the meaning of "macro expansion".

Conceptually, "macro expansion" means evaluating all calls to macros in the source code and replacing them with the return value of this evaluation. Macros take source code as their input and return source code, so by calling them we "expand" the source code, because each call might result in more of it.

For that to work, we need the source code in an accessible form. That's only the case after the parser did its job and we have an AST at hand. So that's why macro expansion happens after the parsing step. And it happens before the evaluation phase, because… well, otherwise it would be too late. There's no point in modifying source code that's not going to be evaluated again.

Translating this to data structures again: where the lexing phase turns strings into tokens and the parsing phase turns tokens into an AST, the macro expansion phase takes the AST, modifies it and returns it before it's evaluated.

Alright! That's the idea behind the macro expansion phase. Now, how are we going to do that? Step by step, because there are two of them.

The first thing we have to do is traverse our AST and find all the macro definitions. A macro definition is nothing more than a let statement in which the value is a macro literal, so we shouldn't have too much trouble with that.

```
let myMacro = macro(x, y) { quote(unquote(x) + unquote(y)); }
```

Once we've found such a macro definition, we have to extract it. That means removing it from the AST and saving it, so we can access it later. The removal is necessary, because otherwise we'd trip over the macros later on in the evaluation phase.

The second step we have to take then is find *the calls to those macros* and evaluate them. That comes pretty close to what we're already doing with function calls in `Eval`. The important difference, as you already know, is that in this phase we don't evaluate the arguments of the call before we evaluate the body. We access them in the macro's body as unevaluated `ast.Nodes`. That's what makes macros different from normal functions; macros work with the unevaluated AST.

Once evaluated we have to reinsert the result of the macro calls into the AST, just like we did with `unquote`, except that now we won't have to convert the return value into an `ast.Node`. Macros already return AST nodes.

Alright! Let's get cracking and start by making and finding macro definitions.

THE MACRO KEYWORD

First things first: in order for us to use the `macro` keyword, we have to teach our lexer about it. That means we have to add a new token type and return the correct token in the lexing process. Let's start with the token type:

```go
// token/token.go

const (
// [...]

    MACRO = "MACRO"
)
```

Now we can add a test to our lexer to make sure that lexing macro literals works as intended:

```go
// lexer/lexer_test.go

func TestNextToken(t *testing.T) {
    input := `let five = 5;
let ten = 10;

let add = fn(x, y) {
  x + y;
};

let result = add(five, ten);
!-/*5;
5 < 10 > 5;

if (5 < 10) {
    return true;
} else {
    return false;
}

10 == 10;
10 != 9;
"foobar"
"foo bar"
[1, 2];
{"foo": "bar"}
macro(x, y) { x + y; };
`

    tests := []struct {
        expectedType    token.TokenType
        expectedLiteral string
    }{
// [...]
        {token.MACRO, "macro"},
        {token.LPAREN, "("},
        {token.IDENT, "x"},
        {token.COMMA, ","},
        {token.IDENT, "y"},
        {token.RPAREN, ")"},
```

```
            {token.LBRACE, "{"},
            {token.IDENT, "x"},
            {token.PLUS, "+"},
            {token.IDENT, "y"},
            {token.SEMICOLON, ";"},
            {token.RBRACE, "}"},
            {token.SEMICOLON, ";"},
            {token.EOF, ""},
        }

// [...]
}
```

The input has been extended with a new line that contains a macro literal, making use of the new macro keyword. It could be reduced to just the macro keyword itself, but I like to have context in test inputs. In the tests themselves the only new token is the one with the token.MACRO type.

```
$ go test ./lexer
--- FAIL: TestNextToken (0.00s)
  lexer_test.go:149: tests[86] - tokentype wrong. expected="MACRO", got="IDENT"
FAIL
FAIL    monkey/lexer    0.007s
```

The test fails. Perfect! Because now we can insert just one carefully crafted line and make it pass:

```
// token/token.go

var keywords = map[string]TokenType{
// [...]
    "macro":  MACRO,
}
```

Ah, yes, there nothing quite like one-line-fixes.

That's it for the lexer. The tests pass. It now knows how to handle the macro keyword in Monkey source code. We can move on to the parser.

Parsing Macro Literals

Now that our lexer knows how to spit out token.MACRO tokens we need to extend our parser so they don't get lost. We need to add support for macro literals.

The test for that looks really similar to the existing one for function literals:

```
// parser/parser_test.go

func TestMacroLiteralParsing(t *testing.T) {
    input := `macro(x, y) { x + y; }`

    l := lexer.New(input)
    p := New(l)
    program := p.ParseProgram()
    checkParserErrors(t, p)

    if len(program.Statements) != 1 {
        t.Fatalf("program.Body does not contain %d statements. got=%d\n",
            1, len(program.Statements))
```

```go
    }

    stmt, ok := program.Statements[0].(*ast.ExpressionStatement)
    if !ok {
        t.Fatalf("statement is not ast.ExpressionStatement. got=%T",
            program.Statements[0])
    }

    macro, ok := stmt.Expression.(*ast.MacroLiteral)
    if !ok {
        t.Fatalf("stmt.Expression is not ast.MacroLiteral. got=%T",
            stmt.Expression)
    }

    if len(macro.Parameters) != 2 {
        t.Fatalf("macro literal parameters wrong. want 2, got=%d\n",
            len(macro.Parameters))
    }

    testLiteralExpression(t, macro.Parameters[0], "x")
    testLiteralExpression(t, macro.Parameters[1], "y")

    if len(macro.Body.Statements) != 1 {
        t.Fatalf("macro.Body.Statements has not 1 statements. got=%d\n",
            len(macro.Body.Statements))
    }

    bodyStmt, ok := macro.Body.Statements[0].(*ast.ExpressionStatement)
    if !ok {
        t.Fatalf("macro body stmt is not ast.ExpressionStatement. got=%T",
            macro.Body.Statements[0])
    }

    testInfixExpression(t, bodyStmt.Expression, "x", "+", "y")
}
```

The test doesn't fail, but won't even compile, because the definition of `ast.MacroLiteral` is missing:

```
$ go test ./parser
# monkey/parser
parser/parser_test.go:958: undefined: ast.MacroLiteral
FAIL    monkey/parser [build failed]
```

That's easily fixed though, since here too we only deviate from the `ast.FunctionLiteral` in name:

```go
// ast/ast.go

type MacroLiteral struct {
    Token      token.Token // The 'macro' token
    Parameters []*Identifier
    Body       *BlockStatement
}

func (ml *MacroLiteral) expressionNode()      {}
func (ml *MacroLiteral) TokenLiteral() string { return ml.Token.Literal }
func (ml *MacroLiteral) String() string {
    var out bytes.Buffer
```

```
    params := []string{}
    for _, p := range ml.Parameters {
        params = append(params, p.String())
    }

    out.WriteString(ml.TokenLiteral())
    out.WriteString("(")
    out.WriteString(strings.Join(params, ", "))
    out.WriteString(") ")
    out.WriteString(ml.Body.String())

    return out.String()
}
```

There is absolutely nothing new here besides the name of the type `MacroLiteral`. Everything else is an exact copy of of `ast.FunctionLiteral`.

But it does the trick. The test now properly blows up, because the parser doesn't know how to turn macro literal tokens into an `*ast.MacroLiteral`:

```
$ go test ./parser
--- FAIL: TestMacroLiteralParsing (0.00s)
  parser_test.go:1124: parser has 6 errors
  parser_test.go:1126: parser error:\
    "no prefix parse function for MACRO found"
  parser_test.go:1126: parser error:\
    "expected next token to be ), got , instead"
  parser_test.go:1126: parser error:\
    "no prefix parse function for , found"
  parser_test.go:1126: parser error:\
    "no prefix parse function for ) found"
  parser_test.go:1126: parser error:\
    "expected next token to be :, got ; instead"
  parser_test.go:1126: parser error:\
    "no prefix parse function for } found"
FAIL
FAIL    monkey/parser   0.008s
```

So far, so good!

In order to make this test pass, we only have to look at how we parse function literals and adapt it to macro literals.

Just like the `fn` keyword, the `macro` keyword can be found (spoken in the terms of our parser) in a prefix position. That means we have to register a new `prefixParseFn` for `token.MACRO` to parse macro literals:

```
// parser/parser.go

func New(l *lexer.Lexer) *Parser {
// [...]

    p.registerPrefix(token.MACRO, p.parseMacroLiteral)

// [...]
}

func (p *Parser) parseMacroLiteral() ast.Expression {
    lit := &ast.MacroLiteral{Token: p.curToken}
```

```go
    if !p.expectPeek(token.LPAREN) {
        return nil
    }

    lit.Parameters = p.parseFunctionParameters()

    if !p.expectPeek(token.LBRACE) {
        return nil
    }

    lit.Body = p.parseBlockStatement()

    return lit
}
```

When the parser now encounters a `macro` keyword it expects to find a pair of `()` following that, containing the parameters of the macro literal. Here we can just reuse the `parseFunctionParameters` method, even though they are macro parameters. We can also reuse `parseBlockStatement` to parse the macro's `Body`, because it's just that: a block statement, containing zero or more statements.

Guess what? The tests pass:

```
$ go test ./parser
ok      monkey/parser   0.008s
```

We can now parse macro literals!

Define Macros

Now that the lexer and the parser know how to build `ast.MacroLiterals`, we can turn our attention to the problem of finding them in the AST. Remember: the first part of the macro expansion phase is extracting all macro definitions from the AST and saving them. In the second part we evaluate them.

As always, we start with a test that defines what we want to happen:

```go
// evaluator/macro_expansion_test.go

package evaluator

import (
    "monkey/ast"
    "monkey/lexer"
    "monkey/object"
    "monkey/parser"
    "testing"
)

func TestDefineMacros(t *testing.T) {
    input := `
    let number = 1;
    let function = fn(x, y) { x + y };
    let mymacro = macro(x, y) { x + y; };
    `

    env := object.NewEnvironment()
```

```go
	program := testParseProgram(input)

	DefineMacros(program, env)

	if len(program.Statements) != 2 {
		t.Fatalf("Wrong number of statements. got=%d",
			len(program.Statements))
	}

	_, ok := env.Get("number")
	if ok {
		t.Fatalf("number should not be defined")
	}
	_, ok = env.Get("function")
	if ok {
		t.Fatalf("function should not be defined")
	}

	obj, ok := env.Get("mymacro")
	if !ok {
		t.Fatalf("macro not in environment.")
	}

	macro, ok := obj.(*object.Macro)
	if !ok {
		t.Fatalf("object is not Macro. got=%T (%+v)", obj, obj)
	}

	if len(macro.Parameters) != 2 {
		t.Fatalf("Wrong number of macro parameters. got=%d",
			len(macro.Parameters))
	}

	if macro.Parameters[0].String() != "x" {
		t.Fatalf("parameter is not 'x'. got=%q", macro.Parameters[0])
	}
	if macro.Parameters[1].String() != "y" {
		t.Fatalf("parameter is not 'y'. got=%q", macro.Parameters[1])
	}

	expectedBody := "(x + y)"

	if macro.Body.String() != expectedBody {
		t.Fatalf("body is not %q. got=%q", expectedBody, macro.Body.String())
	}
}

func testParseProgram(input string) *ast.Program {
	l := lexer.New(input)
	p := parser.New(l)
	return p.ParseProgram()
}
```

With over 50 lines `TestDefineMacros` is quite a mouthful. Thankfully a lot of it is just boilerplate and sanity checks. What it boils down to is making sure that the to-be-written function `DefineMacros` takes a parsed program and an `*object.Environment` as arguments and adds macro definitions from one to the other. It also expects that other let statements are ignored, so they can later be evaluated.

The attentive reader may have anticipated what happens when we try to run this test. Yes, it not only fails, but doesn't even compile. Besides the aforementioned `DefineMacros` function a certain `*object.Macro` is also undefined. Let's fix that first, so we can get closer to a failing test.

Similar to `ast.MacroLiteral` and `ast.FunctionLiteral` the new `object.Macro` is a near exact copy of `object.Function`, except that it has a different name. That makes life easier for us but this next addition not too exciting:

```go
// object/object.go

const (
// [...]
    MACRO_OBJ = "MACRO"
)

type Macro struct {
    Parameters []*ast.Identifier
    Body       *ast.BlockStatement
    Env        *Environment
}

func (m *Macro) Type() ObjectType { return MACRO_OBJ }
func (m *Macro) Inspect() string {
    var out bytes.Buffer

    params := []string{}
    for _, p := range m.Parameters {
        params = append(params, p.String())
    }

    out.WriteString("macro")
    out.WriteString("(")
    out.WriteString(strings.Join(params, ", "))
    out.WriteString(") {\n")
    out.WriteString(m.Body.String())
    out.WriteString("\n}")

    return out.String()
}
```

All the fields and methods are exactly like their counterparts in `object.Function`, only the name of the type itself and the `ObjectType` are different.

And with that the test is finally… Well, it's not passing, or even failing yet, but it now points us in the right direction, I'd say:

```
$ go test ./evaluator
# monkey/evaluator
evaluator/macro_expansion_test.go:21: undefined: DefineMacros
FAIL    monkey/evaluator [build failed]
```

That's a good thing. Because now we can make it compile and *pass* in one swoop by defining `DefineMacro`:

```go
// evaluator/macro_expansion.go

package evaluator

import (
    "monkey/ast"
```

```
    "monkey/object"
)

func DefineMacros(program *ast.Program, env *object.Environment) {
    definitions := []int{}

    for i, statement := range program.Statements {
        if isMacroDefinition(statement) {
            addMacro(statement, env)
            definitions = append(definitions, i)
        }
    }

    for _, i := range definitions {
        program.Statements = append(
            program.Statements[:i],
            program.Statements[i+1:]...,
        )
    }
}
```

This function does two things: finding macro definitions in and removing them from the AST. It finds them by going through all the `program`'s `Statements` and checking each whether it's such a definition with the help of `isMacroDefinition`. If it is, it keeps track of the definition's position in the `Statements` slice so it can remove it at the end.

Of note is that we only allow top-level macro definitions. We don't walk down the `Statements` and check the child nodes for more. The reason for that is the scope of this text. It's not a limitation inherent to the way macros work in Monkey. In fact, the opposite is the case: allowing nested macro definitions might make a fantastic reader exercise, don't you think?

The two helper functions used here, `isMacroDefinition` and `addMacro`, do what their names promise. Here is `isMacroDefinition`:

```
// evaluator/macro_expansion.go

func isMacroDefinition(node ast.Statement) bool {
    letStatement, ok := node.(*ast.LetStatement)
    if !ok {
        return false
    }

    _, ok = letStatement.Value.(*ast.MacroLiteral)
    if !ok {
        return false
    }

    return true
}
```

Yep, a simple check to make sure that we do have a `*ast.LetStatement` at hand that binds a `MacroLiteral` to a name. There is not a lot going on, but this function has a lot of power. It defines what a valid macro definition is and isn't. Consider this:

```
let myMacro = macro(x) { x };
let anotherNameForMyMacro = myMacro;
```

`isMacroDefinition` wouldn't recognize the second let statement as a valid macro definition. That's certainly something to keep in mind.

But if isMacroDefinition returns true, we can pass the let statement to addMacro, which adds the macro definition to the environment:

```go
// evaluator/macro_expansion.go

func addMacro(stmt ast.Statement, env *object.Environment) {
    letStatement, _ := stmt.(*ast.LetStatement)
    macroLiteral, _ := letStatement.Value.(*ast.MacroLiteral)

    macro := &object.Macro{
        Parameters: macroLiteral.Parameters,
        Env:        env,
        Body:       macroLiteral.Body,
    }

    env.Set(letStatement.Name.Value, macro)
}
```

Combined with isMacroDefinition the type assertions in the first two lines are redundant, which is why we ignore possible errors. That's not beautiful, but still the simplest way (for now) to organize both functions. Ignoring that prelude, what addMacro does is adding a newly constructed *object.Macro to the passed in *object.Environment, binding it to the name given in the *ast.LetStatement.

With these three functions defined our test is passing:

```
$ go test ./evaluator
ok      monkey/evaluator        0.009s
```

That means, we are now able to bind macro literals to names in Monkey source code, find them in the AST and save them. Yes, that's pretty neat!

In order to complete the macro expansion phase all that's left for us to do now is to actually expand the macros.

Expand Macros

Before we get started with a test, let's refresh our short term memory: expanding macros means evaluating calls to macros and reinserting the result of the evaluation into the AST, replacing the original call expression.

Does that remind you of something? I thought so. Yes, this is pretty close to how unquote works and you'll see that the implementations are pretty similar. But where an unquote call only causes its single argument to be evaluated, macro calls result in the body of the macro being evaluated, with the arguments made available in the environment.

That being said, here is a test that demonstrates what we want to happen in the macro expansion phase:

```go
// evaluator/macro_expansion.go

func TestExpandMacros(t *testing.T) {
    tests := []struct {
        input    string
        expected string
    }{
        {
            `
```

```
        let infixExpression = macro() { quote(1 + 2); };

        infixExpression();
        `,
        `(1 + 2)`,
    },
    {
        `
        let reverse = macro(a, b) { quote(unquote(b) - unquote(a)); };

        reverse(2 + 2, 10 - 5);
        `,
        `(10 - 5) - (2 + 2)`,
    },
}

for _, tt := range tests {
    expected := testParseProgram(tt.expected)
    program := testParseProgram(tt.input)

    env := object.NewEnvironment()
    DefineMacros(program, env)
    expanded := ExpandMacros(program, env)

    if expanded.String() != expected.String() {
        t.Errorf("not equal. want=%q, got=%q",
            expected.String(), expanded.String())
    }
}
}
```

The basic idea behind these test cases is this: we expand the macro calls in the `input` and compare the result of that expansion against the AST we get from parsing the `expected` source code. In order to do that, we construct a fresh environment, `env`, and use `DefineMacros` to save the macro definitions in the `input` to `env`. Then we use the function we're going to write next, `ExpandMacros`, to expand the macro calls.

It's worth pointing out that the macros in both test cases use `quote` to return a quoted AST node. That's not a random choice, no, that's a rule we now define for our macro system: you *must* return an `*object.Quote` from a macro. If a macro didn't return a quoted AST node, we'd have to convert its return value into one, just like we did when evaluating `unquote` calls with `convertObjectToASTNode`. And that's cumbersome. So instead we just make the usage of `quote` a requirement. Ultimately that makes the macros more powerful, since they're not constrained by what `convertObjectToASTNode` can and can't do.

The first test case, the one defining the `infixExpression` macro, makes sure that macros really return unevaluated source code. The result of a call to `infixExpression` should be the infix expression `1 + 2` and *not* `3`.

The `reverse` macro in the second test case uses more features of the macro system. It has two parameters, `a` and `b`, and returns an infix expression in which the order of the parameters is reversed. The remarkable thing here is, of course, that the parameters won't be evaluated. `2 + 2` doesn't turn into `4` and `10 - 5` doesn't turn into `5`. Instead, `reverse` builds up a new AST node with `quote` and uses `unquote` to access its parameters so it can place them, unevaluated, into a new infix expression. If you're scratching your head about why the calls to `unquote` are necessary: without them the `reverse` macro would simply return `b - a`.

Alright, now that we know how the tests work and what they are supposed to do, how to they fare when passed to `go test`?

```
$ go test ./evaluator
# monkey/evaluator
evaluator/macro_expansion_test.go:95: undefined: ExpandMacros
FAIL    monkey/evaluator [build failed]
```

Not so good, which is good, because now we'll make them pass by defining `ExpandMacros`:

```go
// evaluator/macro_expansion.go

func ExpandMacros(program ast.Node, env *object.Environment) ast.Node {
    return ast.Modify(program, func(node ast.Node) ast.Node {
        callExpression, ok := node.(*ast.CallExpression)
        if !ok {
            return node
        }

        macro, ok := isMacroCall(callExpression, env)
        if !ok {
            return node
        }

        args := quoteArgs(callExpression)
        evalEnv := extendMacroEnv(macro, args)

        evaluated := Eval(macro.Body, evalEnv)

        quote, ok := evaluated.(*object.Quote)
        if !ok {
            panic("we only support returning AST-nodes from macros")
        }

        return quote.Node
    })
}

func isMacroCall(
    exp *ast.CallExpression,
    env *object.Environment,
) (*object.Macro, bool) {
    identifier, ok := exp.Function.(*ast.Identifier)
    if !ok {
        return nil, false
    }

    obj, ok := env.Get(identifier.Value)
    if !ok {
        return nil, false
    }

    macro, ok := obj.(*object.Macro)
    if !ok {
        return nil, false
    }

    return macro, true
}
```

```go
func quoteArgs(exp *ast.CallExpression) []*object.Quote {
    args := []*object.Quote{}

    for _, a := range exp.Arguments {
        args = append(args, &object.Quote{Node: a})
    }

    return args
}

func extendMacroEnv(
    macro *object.Macro,
    args []*object.Quote,
) *object.Environment {
    extended := object.NewEnclosedEnvironment(macro.Env)

    for paramIdx, param := range macro.Parameters {
        extended.Set(param.Value, args[paramIdx])
    }

    return extended
}
```

This is it. That's how we expand macros. That's the complete macro expansion phase in four functions. Let's take a closer look.

`ExpandMacros` uses our trusty helper `ast.Modify` to recursively walk down the `program` AST and find calls to macros. If the `node` at hand is a call expression involving a macro its next step is to evaluate the call.

For that, it takes the arguments and turns them into `*object.Quote`s with the help of `quoteArgs`. Then it uses `extendMacroEnv` to extend the macro's environment with the arguments of the call bound to the parameter names of the macro literal. That's the same preparation that happens when we call a function in `Eval`.

With the arguments quoted and the environment extended, it's time to evaluate the macro. For that `ExpandMacros` uses `Eval` to evaluate the macro's body with the newly extended environment passed in. Finally, and this is important, it returns the quoted AST node, the result of the evaluation. By doing that, instead of modifying a node, it replaces the macro call with the result of the evaluation. It *expands* the macro.

The tests pass:

```
$ go test ./evaluator
ok      monkey/evaluator        0.010s
```

Yes, our macro expansion phase is complete! We have now officially implemented a working macro system for the Monkey programming language! It's time to celebrate and put "meta programmer" on our CVs.

While we may sip champagne now, tradition tells us that must also write a macro called `unless`.

The Mighty Unless Macro

An `unless` macro is usually the first macro shown in any introduction to macros. It's perfect for that: it's easy to understand and to implement and it also demonstrates what a macro system can do and how it does what it does. All the while, it also shows the limitations of normal functions and how macros transcend them, allowing the user to extend a programming language with constructs that look like they're built-in but are "just" macros.

Before we implement it, though, let's see what `unless` is exactly and what it's supposed to do. Consider this piece of Monkey code:

```
if (10 > 5) {
  puts("yes, 10 is greater than 5")
} else {
  puts("holy monkey, 10 is not greater than 5?!")
}
```

This should - one hopes - print `"yes, 10 is greater than 5"`.

Now, if Monkey had `unless` built-in, we could write the above code as follows:

```
unless (10 > 5) {
  puts("holy monkey, 10 is not greater than 5?!")
} else {
  puts("yes, 10 is greater than 5")
}
```

This sometimes makes the code more intention revealing and thus easier to understand. `unless` is a good thing to have.

But we know what adding `unless` to Monkey itself would mean for us: adding a new token type, modifying the lexer, extending the parser with new parsing functions so it can build a new `UnlessExpression` AST node and then adding a new `case` branch to our `Eval` function so it can handle this new node. That's a lot of work.

Here's the great news: now that we have macros in Monkey we don't have to extend Monkey itself; we don't have to change our tokens, lexer, AST, parser or `Eval`; we can implement `unless` as a macro!

```
unless(10 > 5, puts("nope, not greater"), puts("yep, greater"));
// outputs: "yep, greater"
```

This will only print `"yep, greater"`.

Yes, it looks just like a normal function call. The magic lies in how it works. Or: that it works at all. Because if `unless` in the code above were a normal function, the code wouldn't work as expected. Both calls to `puts` would be evaluated before the body of `unless` itself, resulting in both `"nope, not greater"` and `"yep, greater"` being printed. That's not what we want.

As a macro though, `unless` would work exactly like we would expect it to! Let's add it as a test case to our existing `ExpandMacros` function to make sure of it:

```go
// evaluator/macro_expansion_test.go

func TestExpandMacros(t *testing.T) {
	tests := []struct {
		input    string
		expected string
	}{
		// [...]
		{
			`
			let unless = macro(condition, consequence, alternative) {
				quote(if (!(unquote(condition))) {
					unquote(consequence);
```

```
                } else {
                    unquote(alternative);
                });
            };

            unless(10 > 5, puts("not greater"), puts("greater"));
            `,
            `if (!(10 > 5)) { puts("not greater") } else { puts("greater") }`,
        },
    }

    // [...]
}
```

The `unless` macro we define in the test case uses `quote` to construct the AST of an if-conditional, but adds the negating `!` prefix operator and uses `unquote` to insert the three arguments into the AST: `condition`, `consequence` and `alternative`. At the end of the test case we call the newly defined macro to make sure that the AST it produces matches the one we expect.

Now the question is: does the test pass? Does this work? Did we really enhance Monkey in such a way that it allows us to write code that writes code? Using `macro`, `quote` and `unquote`? Yes, we did!

```
$ go test ./evaluator
ok      monkey/evaluator        0.009s
```

It's time we take this on the road.

5.6 - Extending the REPL

The fact that we're able to use macros in test cases is cool. Some would even say, it's amazingly cool. But still… It doesn't feel real until you can use them in the REPL. Thankfully, there is only a tiny number of lines of code stopping us doing amazing macro magic in our Monkey REPL.

Let's add those, shall we?

The first thing we need to add to our REPL is a new, separate environment just for macros:

```go
// repl/repl.go

func Start(in io.Reader, out io.Writer) {
// [...]
  env := object.NewEnvironment()
  macroEnv := object.NewEnvironment()
// [...]
}
```

The existing `env` will be used, just as it was before, by `Eval`. But the new `macroEnv` we will pass to `DefineMacros` and `ExpandMacros`.

Now, since the REPL works on a line-by-line basis, each line is a new `ast.Program`, which we now need to put through the macro expansion phase. So in the main loop of the REPL, just after we parsed a new line and before we pass the `ast.Program` to `Eval`, we can insert our macro expansion phase:

```go
// repl/repl.go
```

```go
func Start(in io.Reader, out io.Writer) {
    // [...]

    for {
        // [...]
        program := p.ParseProgram()
        // [...]

        evaluator.DefineMacros(program, macroEnv)
        expanded := evaluator.ExpandMacros(program, macroEnv)

        evaluated := evaluator.Eval(expanded, env)

        // [...]
    }
}
```

Perfect! That's all it took! We're out of the lab and can hit the road, because we are now able to do macro magic in the REPL!

Due to the way our REPL works, we need to enter the definition of `unless` on one line. But due to the way text works, the line is too long to show here, so I inserted newline breaks, signified by the `\` in it. You can remove those and enter the definition as one line:

```
$ go run main.go
Hello mrnugget! This is the Monkey programming language!
Feel free to type in commands
>> let unless = macro(condition, consequence, alternative)\
 { quote(if (!(unquote(condition))) { unquote(consequence); }\
 else { unquote(alternative); }); };
```

And with that definition entered, we can start to play a drum roll sound in the background and type in - perfectly timed, of course - the following line:

```
>> unless(10 > 5, puts("not greater"), puts("greater"));
greater
```

5.7 - Dream On… In Macros

Our macro system works well and enables us to do some really mind-blowing things. It allows us to write code that writes code. Let's repeat that: it allows us to write code that writes code! That's amazing! We can be proud of ourselves. And the best part? It hasn't even reached its maximum potential. It can be even more powerful, beautiful, elegant and user-friendly. There's still room for improvement.

First on the list of possible improvements is what I like to call the "nasty stuff". Things that are hard to do right but are essential for a production ready system. I know that I mentioned this before, but it bears repeating. I'd also get into trouble if I didn't hit you over the head again with the old "handle yer errors" stick. But the error handling and debugging support in our macro system is severely lacking.

To be precise, we don't have any. Us Monkey programmers, we don't mind living in the fast lane, I know, but sooner or later, when we write some *serious* macros, we'd tear our hair out, because we can't get reliable debugging information about the macro expansion phase. We're also pretty careless about which tokens our modified AST nodes carry around and we didn't even touch the topic of "macro hygiene". I urge you to explore and research these topics.

So there's that. One area where our macro system can be improved; error handling and debugging support. Exhausting to build but essential once we want to get serious about this.

Alright, now that we've talked about the harsh realities of producing a robust and debuggable macro system, let's turn our backs to that and dream for a bit, thinking in terms of "what if…?"

Currently, we only allow passing expressions to `quote` and `unquote`. One consequence of that is that we can't use a return statement or a let statement as an argument in a `quote()` call, for example. The parser won't let us, simply because arguments in call expression can only be of type `ast.Expression`.

But what if we would make `quote` and `unquote` separate keywords and gave them their own AST nodes? That would allows us to extend the parser in such a way that it allows any AST node as arguments to the calls. We could pass in expressions *and* statements! And if we had separate AST nodes, could we extend the allowed syntax even more?

What if we could pass block statements to `quote`/`unquote` calls? That would allow to do something like this:

```
quote() {
  let one = 1;
  let two = 2;
  one + two;
}
```

Wouldn't that be neat?

Now, what if function calls didn't require parentheses around their arguments? What if identifiers could contain special characters? What if we had something like identifiers that resolve to themselves? Like atoms or symbols in other languages. What if every function could take an additional `*ast.BlockStatement` as an argument? What if …?

The point is this: the rules given by the parser determine what constitutes valid Monkey syntax and play a huge part in how expressive and powerful macros can be. When we change these rules, we simultaneously change what macros can and can't do. And there sure are a lot of possible changes to make. Take a look at Elixir or any Lisp, for inspiration, to see how the syntax gives power to the macro systems and how that in turn makes the language itself more powerful and expressive.

The other big influence on the power of our macro system is its ability to access, modify and construct AST nodes. Here's an example. Let's say we have two built-in functions, called `left` and `right`, that respectively return the left and right child nodes of an AST node. That would allow us to do something like this:

```
let plusToMinus = macro(infixExpression) {
  quote(unquote(left(infixExpression)) - unquote(right(infixExpression)))
}
```

Now *that* would enable us to write some really, really interesting macros!

What if we had more of these functions? Something like an `operator` function, that returns the operator of an infix expression? Or an `arguments` function that returns an array of the argument nodes in a call expression? Or a generic `children` function? What if our built-in functions `len`, `first` and `last` would work with AST nodes?

Now, here is the ultimate "what if" of them all: what if the AST was built with the same data structures that the rest of the language uses? Imagine for a second that the Monkey AST was built purely out of `object.Array`, `object.Hash`, `object.String`, `object.Integer`, and others. Just imagine what that would enable us to do and seamless the whole experience would be. Inspiring, right? If you want to get a taste of that, take a look at a Lisp like Clojure, Racket or Guile, or non-Lisp languages with great macro systems like Elixir and Julia.

So, you see, there's a lot of room for dreams when writing code that writes code.