THORSTEN BALL

WRITING AN INTERPRETER IN GO

WRITING AN INTERPRETER IN GO

THORSTEN BALL

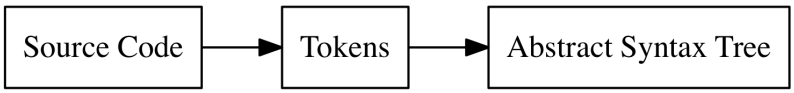
CHAPTER 1

LEXING

1.1 - LEXICAL ANALYSIS

In order for us to work with source code we need to turn it into a more accessible form. As easy as plain text is to work with in our editor, it becomes cumbersome pretty fast when trying to interpret it in a programming language as another programming language.

So, what we need to do is represent our source code in other forms that **are** easier to work with. We're going to change the representation of our source code two times before we evaluate it:



The first transformation, from source code to tokens, is called "lexical analysis", or "lexing" for short. It's done by a lexer (also called tokenizer or scanner – some use one word or the other to denote subtle differences in behaviour, which we can ignore in this book).

Tokens itself are small, easily categorizable data structures that are then fed to the parser, which does the second transformation and turns the tokens into an "Abstract Syntax Tree".

Here's an example. This is the input one gives to a lexer:

```
"let x = 5 + 5;"
```

And what comes out of the lexer looks kinda like this:

```
[
  LET,
  IDENTIFIER("x"),
  EQUAL_SIGN,
  INTEGER(5),
  PLUS_SIGN,
  INTEGER(5),
  SEMICOLON
]
```

All of these tokens have the original source code representation attached. `"let"` in the case of `LET`, `"+"` in the case of `PLUS_SIGN`, and so on. Some, like `IDENTIFIER` and `INTEGER` in our example,

also have the concrete values they represent attached: `5` (not `"5"`!) in the case of `INTEGER` and `"x"` in the case of `IDENTIFIER`. But what exactly constitutes a "token" varies between different lexer implementations. As an example, some lexers only convert the `"5"` to an integer in the parsing stage, or even later, and not when constructing tokens.

A thing to note about this example: whitespace characters don't show up as tokens. In our case that's okay, because whitespace length is not significant in the Monkey language. Whitespace merely acts as a separator for other tokens. It doesn't matter if we type this:

```
let x = 5;
```

Or if we type this:

```
let   x   =   5;
```

In other languages, like Python, the length of whitespace *is* significant. That means the lexer can't just "eat up" the whitespace and newline characters. It has to output the whitespace characters as tokens so the parser can later on make sense of them (or output an error, of course, if there are not enough or too many).

A production-ready lexer might also attach the line number, column number and filename to a token. Why? For example, to later output more useful error messages in the parsing stage. Instead of `"error: expected semicolon token"` it can output:

```
"error: expected semicolon token. line 42, column 23, program.monkey"
```

We're not going to bother with that. Not because it's too complex, but because it would take away from the essential simpleness of the tokens and the lexer, making it harder to understand.

1.2 - DEFINING OUR TOKENS

The first thing we have to do is to define the tokens our lexer is going to output. We're going to start with just a few token definitions and then add more when extending the lexer.

The subset of the Monkey language we're going to lex in our first step looks like this:

```
let five = 5;
let ten = 10;

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

let result = add(five, ten);
```

Let's break this down: which types of tokens does this example contain? First of all, there are the numbers like `5` and `10`. These are pretty obvious. Then we have the variable names `x`, `y`, `add` and `result`. And then there are also these parts of the language that are not numbers, just words, but no variable names either, like `let` and `fn`. Of course, there are also a lot of special characters: `(`, `)`, `{`, `}`, `=`, `,`, `;`.

The numbers are just integers and we're going to treat them as such and give them a separate type. In the lexer or parser we don't care if the number is `5` or `10`, we just want to know if it's a number. The same goes for "variable names": we'll call them "identifiers" and treat them the same. Now, the other words, the ones that look like identifiers, but aren't really identifiers, since they're part of the language, are called "keywords". We won't group these together since it **should** make a difference in the parsing stage whether we encounter a `let` or a `fn`. The

same goes for the last category we identified: the special characters. We'll treat each of them separately, since it is a big difference whether or not we have a `(` or a `)` in the source code.

Let's define our `Token` data structure. Which fields does it need? As we just saw, we definitely need a "type" attribute, so we can distinguish between "integers" and "right bracket" for example. And it also needs a field that holds the literal value of the token, so we can reuse it later and the information whether a "number" token is a `5` or a `10` doesn't get lost.

In a new `token` package we define our `Token` struct and our `TokenType` type:

```
// token/token.go

package token

type TokenType string

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

We defined the `TokenType` type to be a `string`. That allows us to use many different values as `TokenTypes`, which in turn allows us to distinguish between different types of tokens. Using `string` also has the advantage of being easy to debug without a lot of boilerplate and helper functions: we can just print a `string`. Of course, using a `string` might not lead to the same performance as using an `int` or a `byte` would, but for this book a `string` is perfect.

As we just saw, there is a limited number of different token types in the Monkey language. That means we can define the possible `TokenTypes` as constants. In the same file we add this:

```
// token/token.go

const (
    ILLEGAL = "ILLEGAL"
    EOF     = "EOF"

    // Identifiers + literals
    IDENT = "IDENT" // add, foobar, x, y, ...
    INT   = "INT"   // 1343456

    // Operators
    ASSIGN   = "="
    PLUS     = "+"

    // Delimiters
    COMMA     = ","
    SEMICOLON = ";"

    LPAREN = "("
    RPAREN = ")"
    LBRACE = "{"
    RBRACE = "}"

    // Keywords
    FUNCTION = "FUNCTION"
    LET      = "LET"
)
```

As you can see there are two special types: `ILLEGAL` and `EOF`. We didn't see them in the example

above, but we'll need them. `ILLEGAL` signifies a token/character we don't know about and `EOF` stands for "end of file", which tells our parser later on that it can stop.

So far so good! We are ready to start writing our lexer.

1.3 - THE LEXER

Before we start to write code, let's be clear about the goal of this section. We're going to write our own lexer. It will take source code as input and output the tokens that represent the source code. It will go through its input and output the next token it recognizes. It doesn't need to buffer or save tokens, since there will only be one method called `NextToken()`, which will output the next token.

That means, we'll initialize the lexer with our source code and then repeatedly call `NextToken()` on it to go through the source code, token by token, character by character. We'll also make life simpler here by using `string` as the type for our source code. Again: in a production environment it makes sense to attach filenames and line numbers to tokens, to better track down lexing and parsing errors. So it would be better to initialize the lexer with an `io.Reader` and the filename. But since that would add more complexity we're not here to handle, we'll start small and just use a `string` and ignore filenames and line numbers.

Having thought this through, we now realize that what our lexer needs to do is pretty clear. So let's create a new package and add a first test that we can continuously run to get feedback about the working state of the lexer. We're starting small here and will extend the test case as we add more capabilities to the lexer:

```
// lexer/lexer_test.go

package lexer

import (
    "testing"

    "monkey/token"
)

func TestNextToken(t *testing.T) {
    input := `=+(){},;`

    tests := []struct {
        expectedType    token.TokenType
        expectedLiteral string
    }{
        {token.ASSIGN, "="},
        {token.PLUS, "+"},
        {token.LPAREN, "("},
        {token.RPAREN, ")"},
        {token.LBRACE, "{"},
        {token.RBRACE, "}"},
        {token.COMMA, ","},
        {token.SEMICOLON, ";"},
        {token.EOF, ""},
    }

    l := New(input)
```

```
    for i, tt := range tests {
        tok := l.NextToken()

        if tok.Type != tt.expectedType {
            t.Fatalf("tests[%d] - tokentype wrong. expected=%q, got=%q",
                i, tt.expectedType, tok.Type)
        }

        if tok.Literal != tt.expectedLiteral {
            t.Fatalf("tests[%d] - literal wrong. expected=%q, got=%q",
                i, tt.expectedLiteral, tok.Literal)
        }
    }
}
```

Of the course, the tests fail – we haven't written any code yet:

```
$ go test ./lexer
# monkey/lexer
lexer/lexer_test.go:27: undefined: New
FAIL    monkey/lexer [build failed]
```

So let's start by defining the `New()` function that returns `*Lexer`.

```
// lexer/lexer.go
package lexer

type Lexer struct {
    input        string
    position     int  // current position in input (points to current char)
    readPosition int  // current reading position in input (after current char)
    ch           byte // current char under examination
}

func New(input string) *Lexer {
    l := &Lexer{input: input}
    return l
}
```

Most of the fields in `Lexer` are pretty self-explanatory. The ones that might cause some confusion right now are `position` and `readPosition`. Both will be used to access characters in `input` by using them as an index, e.g.: `l.input[l.readPosition]`. The reason for these two "pointers" pointing into our input string is the fact that we will need to be able to "peek" further into the input and look after the current character to see what comes up next. `readPosition` always points to the "next" character in the input. `position` points to the character in the input that corresponds to the `ch` byte.

A first helper method called `readChar()` should make the usage of these fields easier to understand:

```
// lexer/lexer.go

func (l *Lexer) readChar() {
    if l.readPosition >= len(l.input) {
        l.ch = 0
    } else {
        l.ch = l.input[l.readPosition]
    }
    l.position = l.readPosition
```

```
    l.readPosition += 1
}
```

The purpose of `readChar` is to give us the next character and advance our position in the `input` string. The first thing it does is to check whether we have reached the end of `input`. If that's the case it sets `l.ch` to `0`, which is the ASCII code for the `"NUL"` character and signifies either "we haven't read anything yet" or "end of file" for us. But if we haven't reached the end of `input` yet it sets `l.ch` to the next character by accessing `l.input[l.readPosition]`.

After that `l.position` is updated to the just used `l.readPosition` and `l.readPosition` is incremented by one. That way, `l.readPosition` always points to the next position where we're going to read from next and `l.position` always points to the position where we last read. This will come in handy soon enough.

While talking about `readChar` it's worth pointing out that the lexer only supports ASCII characters instead of the full Unicode range. Why? Because this lets us keep things simple and concentrate on the essential parts of our interpreter. In order to fully support Unicode and UTF-8 we would need to change `l.ch` from a `byte` to `rune` and change the way we read the next characters, since they could be multiple bytes wide now. Using `l.input[l.readPosition]` wouldn't work anymore. And then we'd also need to change a few other methods and functions we'll see later on. So it's left as an exercise to the reader to fully support Unicode (and emojis!) in Monkey.

Let's use `readChar` in our `New()` function so our `*Lexer` is in a fully working state before anyone calls `NextToken()`, with `l.ch`, `l.position` and `l.readPosition` already initialized:

```
// lexer/lexer.go

func New(input string) *Lexer {
    l := &Lexer{input: input}
    l.readChar()
    return l
}
```

Our tests now tell us that calling `New(input)` doesn't result in problems anymore, but the `NextToken()` method is still missing. Let's fix that by adding a first version:

```
// lexer/lexer.go

import "monkey/token"

func (l *Lexer) NextToken() token.Token {
    var tok token.Token

    switch l.ch {
    case '=':
        tok = newToken(token.ASSIGN, l.ch)
    case ';':
        tok = newToken(token.SEMICOLON, l.ch)
    case '(':
        tok = newToken(token.LPAREN, l.ch)
    case ')':
        tok = newToken(token.RPAREN, l.ch)
    case ',':
        tok = newToken(token.COMMA, l.ch)
    case '+':
        tok = newToken(token.PLUS, l.ch)
    case '{':
```

```
        tok = newToken(token.LBRACE, l.ch)
    case '}':
        tok = newToken(token.RBRACE, l.ch)
    case 0:
        tok.Literal = ""
        tok.Type = token.EOF
    }

    l.readChar()
    return tok
}

func newToken(tokenType token.TokenType, ch byte) token.Token {
    return token.Token{Type: tokenType, Literal: string(ch)}
}
```

That's the basic structure of the `NextToken()` method. We look at the current character under examination (`l.ch`) and return a token depending on which character it is. Before returning the token we advance our pointers into the input so when we call `NextToken()` again the `l.ch` field is already updated. A small function called `newToken` helps us with initializing these tokens.

Running the tests we can see that they pass:

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

Great! Let's now extend the test case so it starts to resemble Monkey source code.

```
// 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);
`

    tests := []struct {
        expectedType    token.TokenType
        expectedLiteral string
    }{
        {token.LET, "let"},
        {token.IDENT, "five"},
        {token.ASSIGN, "="},
        {token.INT, "5"},
        {token.SEMICOLON, ";"},
        {token.LET, "let"},
        {token.IDENT, "ten"},
        {token.ASSIGN, "="},
        {token.INT, "10"},
        {token.SEMICOLON, ";"},
        {token.LET, "let"},
        {token.IDENT, "add"},
        {token.ASSIGN, "="},
        {token.FUNCTION, "fn"},
```

```
            {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.LET, "let"},
            {token.IDENT, "result"},
            {token.ASSIGN, "="},
            {token.IDENT, "add"},
            {token.LPAREN, "("},
            {token.IDENT, "five"},
            {token.COMMA, ","},
            {token.IDENT, "ten"},
            {token.RPAREN, ")"},
            {token.SEMICOLON, ";"},
            {token.EOF, ""},
        }
    // [...]
    }
```

Most notably the `input` in this test case has changed. It looks like a subset of the Monkey language. It contains all the symbols we already successfully turned into tokens, but also new things that are now causing our tests to fail: identifiers, keywords and numbers.

Let's start with the identifiers and keywords. What our lexer needs to do is recognize whether the current character is a letter and if so, it needs to read the rest of the identifier/keyword until it encounters a non-letter-character. Having read that identifier/keyword, we then need to find out if it is a identifier or a keyword, so we can use the correct `token.TokenType`. The first step is extending our switch statement:

```
// lexer/lexer.go

import "monkey/token"

func (l *Lexer) NextToken() token.Token {
    var tok token.Token

    switch l.ch {
// [...]
    default:
        if isLetter(l.ch) {
            tok.Literal = l.readIdentifier()
            return tok
        } else {
            tok = newToken(token.ILLEGAL, l.ch)
        }
    }
// [...]
}

func (l *Lexer) readIdentifier() string {
```

```
    position := l.position
    for isLetter(l.ch) {
        l.readChar()
    }
    return l.input[position:l.position]
}

func isLetter(ch byte) bool {
    return 'a' <= ch && ch <= 'z' || 'A' <= ch && ch <= 'Z' || ch == '_'
}
```

We added a `default` branch to our switch statement, so we can check for identifiers whenever the `l.ch` is not one of the recognized characters. We also added the generation of `token.ILLEGAL` tokens. If we end up there, we truly don't know how to handle the current character and declare it as `token.ILLEGAL`.

The `isLetter` helper function just checks whether the given argument is a letter. That sounds easy enough, but what's noteworthy about `isLetter` is that changing this function has a larger impact on the language our interpreter will be able to parse than one would expect from such a small function. As you can see, in our case it contains the check `ch == '_'`, which means that we'll treat _ as a letter and allow it in identifiers and keywords. That means we can use variable names like `foo_bar`. Other programming languages even allow `!` and `?` in identifiers. If you want to allow that too, this is the place to sneak it in.

`readIdentifier()` does exactly what its name suggests: it reads in an identifier and advances our lexer's positions until it encounters a non-letter-character.

In the `default:` branch of the switch statement we use `readIdentifier()` to set the `Literal` field of our current token. But what about its `Type`? Now that we have read identifiers like `let`, `fn` or `foobar`, we need to be able to tell user-defined identifiers apart from language keywords. We need a function that returns the correct `TokenType` for the token literal we have. What better place than the `token` package to add such a function?

```
// token/token.go

var keywords = map[string]TokenType{
    "fn":  FUNCTION,
    "let": LET,
}

func LookupIdent(ident string) TokenType {
    if tok, ok := keywords[ident]; ok {
        return tok
    }
    return IDENT
}
```

`LookupIdent` checks the `keywords` table to see whether the given identifier is in fact a keyword. If it is, it returns the keyword's `TokenType` constant. If it isn't, we just get back `token.IDENT`, which is the `TokenType` for all user-defined identifiers.

With this in hand we can now complete the lexing of identifiers and keywords:

```
// lexer/lexer.go

func (l *Lexer) NextToken() token.Token {
    var tok token.Token
```

```
    switch l.ch {
// [...]
    default:
        if isLetter(l.ch) {
            tok.Literal = l.readIdentifier()
            tok.Type = token.LookupIdent(tok.Literal)
            return tok
        } else {
            tok = newToken(token.ILLEGAL, l.ch)
        }
    }
// [...]
}
```

The early exit here, our `return tok` statement, is necessary because when calling `readIdentifier()`, we call `readChar()` repeatedly and advance our `readPosition` and `position` fields past the last character of the current identifier. So we don't need the call to `readChar()` after the switch statement again.

Running our tests now, we can see that `let` is identified correctly but the tests still fail:

```
$ go test ./lexer
--- FAIL: TestNextToken (0.00s)
  lexer_test.go:70: tests[1] - tokentype wrong. expected="IDENT", got="ILLEGAL"
FAIL
FAIL    monkey/lexer 0.008s
```

The problem is the next token we want: a `IDENT` token with `"five"` in its `Literal` field. Instead we get an `ILLEGAL` token. Why is that? Because of the whitespace character between "let" and "five". But in Monkey whitespace only acts as a separator of tokens and doesn't have meaning, so we need to skip over it entirely:

```
// lexer/lexer.go

func (l *Lexer) NextToken() token.Token {
    var tok token.Token

    l.skipWhitespace()

    switch l.ch {
// [...]
}

func (l *Lexer) skipWhitespace() {
    for l.ch == ' ' || l.ch == '\t' || l.ch == '\n' || l.ch == '\r' {
        l.readChar()
    }
}
```

This little helper function is found in a lot of parsers. Sometimes it's called `eatWhitespace` and sometimes `consumeWhitespace` and sometimes something entirely different. Which characters these functions actually skip depends on the language being lexed. Some language implementations do create tokens for newline characters for example and throw parsing errors if they are not at the correct place in the stream of tokens. We skip over newline characters to make the parsing step later on a little easier.

With `skipWhitespace()` in place, the lexer trips over the `5` in the `let five = 5;` part of our test input. And that's right, it doesn't know yet how to turn numbers into tokens. It's time to add this.

As we did previously for identifiers, we now need to add more functionality to the `default` branch of our switch statement.

```
// lexer/lexer.go

func (l *Lexer) NextToken() token.Token {
    var tok token.Token

    l.skipWhitespace()

    switch l.ch {
// [...]
    default:
        if isLetter(l.ch) {
            tok.Literal = l.readIdentifier()
            tok.Type = token.LookupIdent(tok.Literal)
            return tok
        } else if isDigit(l.ch) {
            tok.Type = token.INT
            tok.Literal = l.readNumber()
            return tok
        } else {
            tok = newToken(token.ILLEGAL, l.ch)
        }
    }
// [...]
}

func (l *Lexer) readNumber() string {
    position := l.position
    for isDigit(l.ch) {
        l.readChar()
    }
    return l.input[position:l.position]
}

func isDigit(ch byte) bool {
    return '0' <= ch && ch <= '9'
}
```

As you can see, the added code closely mirrors the part concerned with reading identifiers and keywords. The `readNumber` method is exactly the same as `readIdentifier` except for its usage of `isDigit` instead of `isLetter`. We could probably generalize this by passing in the character-identifying functions as arguments, but won't, for simplicity's sake and ease of understanding.

The `isDigit` function is as simple as `isLetter`. It just returns whether the passed in byte is a Latin digit between `0` and `9`.

With this added, our tests pass:

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

I don't know if you noticed, but we simplified things a lot in `readNumber`. We only read in *integers*. What about floats? Or numbers in hex notation? Octal notation? We ignore them and just say that Monkey doesn't support this. Of course, the reason for this is again the educational aim and limited scope of this book.

It's time to pop the champagne and celebrate: we successfully turned the small subset of the

Monkey language we used in the our test case into tokens!

With this victory under our belt, it's easy to extend the lexer so it can tokenize a lot more of Monkey source code.

END OF SAMPLE

You've reached the end of the sample. I hope you enjoyed it. You can buy the full version of the book (in multiple formats, including the code) online at:

https://interpreterbook.com