# Challenges

## Chapter 1: Introduction

### Challenge 1.1

The six domain-specific languages are:

- Markdown: A lightweight markup language with plain-text-formatting syntax
- Hypertext Markup Language (HTML): Standard markup langauge for documents designed to be displayed in a web browser
- gitignore: A gitignore file tells Git which files to ignore
- Makefile: Is used by the build management tool *make* to describe the build process
- YAML Ain't Markup Language (YAML): Simplified markup language for data serialization
- Syntactically Awesome Style Sheets (Sass): Preprocessor scripting language that is interpreted or compiled into Cascading Style Sheets (CSS)
- ...

### Challenge 1.2

See "HelloWorld.java"

### Challenge 1.3

See "/c_programming/doubly_linked_list/" for all the files. I did it with integers instead of strings though...

## Chapter 2: A Map of the Territory

### Challenge 2.1

// TODO

### Challenge 2.2

// TODO

### Challenge 2.3

// TODO

## Chapter 3: The Lox Language

### Challenge 3.1

See "./lox_programs/".

### Challenge 3.2

It is kind of hard for me to come up with questions about the language. See the lox programs and the comments I wrote. There I documented some "special" things I noticed. Also under Challenge 3.3 I listed some thing that came to my mind.

### Challenge 3.3

Missing features for "real" programs:

- Multiple inheritance
- Type casting
- ++ and -- shortcuts for incrementing or decrementing
- No enumeration type
- No overloading
- Inner classes
- No Arrays
- No possibility to format the output of the *print* statement

## Chapter 4: Scanning

### Challenge 4.1

In regular grammars it is allowed only to have one non-terminal on the left side of a production rule. On the right only one terminal and a maximum of one non-terminal symbol can be used. E.g. A --> aB or A --> a. The grammars of Python and Haskell break with this rule. In fact a lot of programming languages count on context-free (Type 2) grammars where the right side of a production rule can contain an arbitrary sequence of terminals and non-terminals. Such grammar can be described by the Backus-Naur-Form (BNF). Context-free grammars are used because they are more powerful than regular (Type 3) languages/grammars. E.g. with a type 3 grammar you cannot even specify well formed bracket expressions. Type 1 grammars on the other hand are too complex. Instead the context is inferred at a leater state in the parser.
https://qr.ae/pvegwT

### Challenge 4.2

Spaces in CoffeeScript:

- To format the content of block strings delimited by """ or '''
- Multi line strings are joined by a single space
- Objects can be defined in YAML syntax which also recognizes spaces
- For distinguishing operators from XML tags in JSX
- Using Markdown syntax in *Literate CoffeeScript*

https://coffeescript.org/

Spaces in Ruby:

- In line oriented string literals there must be no space between << and the terminator
- In case of self assignment the syntax *expr op= expr* does not allow whitespace between *op* and =
- For creating arrays of strings with %w expressions

https://ruby-doc.org/docs/ruby-doc-bundle/Manual/man-1.4/syntax.html

Spaces in the C preprocessor:
The preprocessor in some cases inserts whitespace elements in its token stream to eliminate ambiguity.

https://stackoverflow.com/questions/37796947/spaces-inserted-by-the-c-preprocessor

## Challenge 4.3

- In order to do stuff like in challenge 4.2
- If the parser needs to know about whitespaces
- If whitespaces change the semantic of a program

## Challenge 4.4

Solution without nesting:

```
private void blockComment()
{
    while (!isAtEnd())
    {
        if (peek() == '*' && peekNext() == '/')
        {
            // The block ends and we need to consume "*" and "/".
            advance();
            advance();
            return;
        }
        advance();
    }
    Lox.error(line, "Block comment not terminated.");
}
```

```
case '/':
    if (match('/'))
    {
        // A comment goes unitl the end of the line.
        while (peek() != '\n' && !isAtEnd())
            advance();
    }
    else if (match('*'))
    {
        blockComment();
    }
    else
    {
        addToken(SLASH);
    }
    break;
```

Solution with nesting:

```
private void blockComment()
{
    while (!isAtEnd())
    {
        // There is a nested block comment.
        if (peek() == '/' && peekNext() == '*')
        {
            advance();
            advance();
            blockComment();
        }

        if (peek() == '*' && peekNext() == '/')
        {
            // The block ends and we need to consume "*" and "/".
            advance();
            advance();
            return;
        }

        if (!isAtEnd() && advance() == '\n')
        {
            line++;
        }
    }
    Lox.error(line, "Block comment not terminated.");
}
```

The problem with this solution is that things like

```
/*
    This is a comment
    /*
        This is a nested comment
    */
*/
```

work and

```
/*
    This is a comment
    /*
        This is a nested comment

*/
```

is also regognized as an error but in the below example the tokens *STAR* and *SLASH* are scanned 😦.

```
/*
    This is a comment
    */
*/
```

# Chapter 5: Representing Code

## Challenge 5.1

Given grammar:

```
expr -> expr ( "(" ( expr ( "," expr )* )? ")" | "." IDENTIFIER )+
      | IDENTIFIER
      | NUMBER
```

This grammar produces stuff like *IDENTIFIER.IDENTIFIER*, *NUMBER.IDENTIFIER.IDENTIFIER*, *IDENTIFIER(IDENTIFIER, NUMBER)* or IDENTIFIER(). Without the syntactic sugar I think the grammar should look something like this:

```
expr -> expr call;
expr -> IDENTIFIER;
expr -> NUMBER;
call -> "." IDENTIFIER;
call -> "." IDENTIFIER call;
call -> "(" ")";
call -> "(" expr ")";
call -> "(" expr argument ")";
argument -> "," expr;
argument -> "," expr argument;
```

The kind of expressions described by that might be class instantiation, function calls or accessing variables of a class instance.

## Challenge 5.2

I do not know any functional language well enough to devise such a pattern I think...

## Challenge 5.3

See *com.craftinginterpreters.lox.RpnRpinter* class. Question that came to mind was how to parse unary operators in RPN. The answer I found stated that you just do not: https://stackoverflow.com/questions/64867998/how-do-unary-operators-get-parsed-using-rpn.

# Chapter 6: Parsing Expressions

## Challenge 6.1

Comma operator has lowest precedence. Change grammar accordingly:

```
expression  -> comma ;
comma       -> ternary ( "," comma )* ;
ternary     -> equality "?" equality ":" ternary ;
equality    -> comparison ( ( "==" | "!=" ) comparison )* ;
comparison  -> term ( ( ">=" | ">" | "<" | "<=" ) term )* ;
term        -> factor ( ( "+" | "-" ) factor )* ;
factor      -> unary ( ("/" | "*" ) unary )* ;
unary       -> ( "!" | "-" ) unary | primary ;
primary     -> NUMBER | STRING | "true" | "false" | "nil" | "(" expression ")" ;
```

In Java:

```
private Expr comma()
{
    Expr expr = ternary();
    if (match(COMMA))
    {
        List<Expr> exprs = new ArrayList<>();
        exprs.add(expr);
        exprs.add(comma());
        while (match(COMMA))
        {
            exprs.add(comma());
        }
        return new Expr.Comma(exprs);
    }

    return expr;
}
```

Comma operator allows things like

```
int a = 2, b = 3; // Not the comma operator!
int i = (a = a + 2, b + 1); // a will be incremented by 2 and i will have the
value b + 1!
```

Infos:
https://en.cppreference.com/w/c/language/operator_precedence
https://en.wikipedia.org/wiki/Comma_operator

## Challenge 6.2

Wanted: 4 == 5 ? expression : expression;
The expression between the `?` and `:` is treated like a grouped expression. The whole expression is right-

associative.

Add the ternary operator to the grammar:

```
expression  -> ternary ;
ternary     -> equality "?" equality ":" ternary ;
equality    -> comparison ( ( "==" | "!=" ) comparison )* ;
comparison  -> term ( ( ">=" | ">" | "<" | "<=" ) term )* ;
term        -> factor ( ( "+" | "-" ) factor )* ;
factor      -> unary ( ("/" | "*" ) unary )* ;
unary       -> ( "!" | "-" ) unary | primary ;
primary     -> NUMBER | STRING | "true" | "false" | "nil" | "(" expression ")" ;
```

Program it into the parser:

```
private Expr ternary()
{
    Expr expr = equality();
    if (match(QUESTION_MARK))
    {
        Expr inner = equality();
        consume(COLON, "Expected ':' for ternary operator.");
        Expr right = ternary();
        expr = new Expr.Ternary(expr, inner, right);
    }

    return expr;
}
```

## Challenge 6.3

Add error productions to grammar:

```
expression  -> comma ;
comma       -> ternary ( "," comma )* ;
ternary     -> equality "?" equality ":" ternary ;
equality    -> comparison? ( ( "==" | "!=" ) comparison )* ;
comparison  -> term? ( ( ">=" | ">" | "<" | "<=" ) term )* ;
term        -> factor? ( ( "+" | "-" ) factor )* ;
factor      -> unary? ( ( "/" | "*" ) unary )* ;
unary       -> ( "!" | "-" | "+" ) unary | primary ;
primary     -> NUMBER | STRING | "true" | "false" | "nil" | "(" expression ")" ;
```

Error if unary '+' expression:

```
private Expr unary()
{
    if (match(PLUS))
    {
        error(peek(), "Unary '+' expressions are not supported.");
    }
```

# Chapter 7: Evaluating Expressions

## Challenge 7.1

My first intuition is that this feature should not be implemented. The reason is that many decisions have to be made about what things like `3 < "pumpkin"` will evaluate to. Those decisions are not transparent to the user and might lead to confusion. Java does not overload the comparison operators for strings and numbers (one can make use of the `Comparable` interface or `Comparator` class). JavaScript allows for comparing different types. The rules are actually not too complex.

- JavaScript converts the string of the expression to a number
- If its not a number (cannot be converted) the value is just `NaN`
- If its the empty string the value is 0
- After conversion the comparison can continue normally

I think those rules are not too bad but also not super useful if you can instead just explicitly convert the type...

## Challenge 7.2

Enable concatenation of string and number:

```
case PLUS:
    if (left instanceof Double && right instanceof Double)
        return (double)left + (double)right;
    if (left instanceof String && right instanceof String)
        return (String)left + (String)right;
    if (left instanceof String && right instanceof Double)
        return (String)left + String.valueOf(right);
    if (left instanceof Double && right instanceof String)
        return String.valueOf(left) + (String)right;
    throw new RuntimeError(expr.operator, "Operands must be two numbers or two
strings.");
```

## Challenge 7.3

Right now division by zero looks like this: `x / 0 = Infinity`. If this happens inside a greater term the term will still evaluate to `Infinity`. JavaScript also does it like this. Java throws an `ArithmeticException`. I think it is best to handle division by zero similar to java and terminate the program because if an expression returns `Infinity` the programmer probably did something wrong and you cannot use the value of the expression

afterwards.
Handle division by zero:

```
case SLASH:
    checkNumberOperands(expr.operator, left, right);
    if ((double)right == 0)
    {
        throw new RuntimeError(expr.operator, "Division by zero.");
    }
    return (double)left / (double)right;
```

# Chapter 8: Statements and State

## Challenge 8.1

In `Lox.runPrompt(String)` check if the line ends with a smicolon. If not it cannot be a statement so we evaluate it. To be able to parse the entered string we add the semicolon.

```
if (!line.endsWith(";"))
    // The line is not a statement.
    evaluate(line + ";");
```

Now we can continue as in `Lox.run(String)`. The parsed statement must be a statement expression. From this statement we can get the encapsulated expression and interpret its value to print it on the screen.
Finished...

```
private static void evaluate(String source)
{
    Scanner scanner = new Scanner(source);
    List<Token> tokens = scanner.scanTokens();
    Parser parser = new Parser(tokens);
    List<Stmt> statements = parser.parse();

    if (hadError || !(statements.get(0) instanceof Stmt.Expression))
        return;

    System.out.println(((Stmt.Expression)statements.get(0))
                       .expression.accept(new Interpreter()));
}
```

## Challenge 8.2

Challenge:

```
// Challenge 8.2
// No initializers.
var a;
var b;

a = "assigned";
print a; // OK, was assigned first.

print b; // Should result in error instead of "nil".
```

Solution: Add additional condition that `null` is not a valid value in `Environment.get(Object)`:

```
Object get(Token name)
{
    if (values.containsKey(name.lexeme) && values.get(name.lexeme) != null)
        return values.get(name.lexeme);

    if (enclosing != null)
        return enclosing.get(name);

    throw new RuntimeError(name, "Undefined variable '" + name.lexeme + "'.");
}
```

### Challenge 8.3

Expectation: First a `Stmt.Var` ist parsed. Afterwards a block will be parsed with another `Stmt.Var` inside and a `print` statement. When interpreted `a` will be assigned the value "1" in the most outer environment. Then another envoironment is opened inside of it because of the upcoming block statement. Inside this block we have another variable declaration. First the initializer is interpreted. In the initializer we refer to the `a` variable of the outer scope so the result of this expression should be "1 + 2 = 3". This value is assigned to the "new" `a` and gets printed. So the expected answer is that "3" is printed but outside of the block `a` has still the value "1". This would not be the case if we did not search for a refered variable in the most inner scope first.

Java gives an error "Duplicate local variable" in this case.

## Chapter 9: Control Flow

### Challenge 9.1

First class functions: A language has first class functions if functions are treated as any other variable.

```
// First class functions example
fun hello(functionArg)
{
    print "Hello";
    print functionArg();
}
```

```
fun world()
{
    return "World";
}

hello(world);
```

With first class functions one can execute different code based on which function is passed as an argument. This can also be used for branching inside the code.

Dynamic dispatch means determining which function to call based on the objects type.

```
// Dynamic dispatch example
class Pet
{
    speak()
    {
        return "Undefined";
    }
}

class Dog < Pet
{
    speak()
    {
        return "Woof";
    }
}

class Cat < Pet
{
    speak()
    {
        return "Cat";
    }
}

fun speak(pet)
{
    print pet.speak();
}

var pet = Pet();
var dog = Dog();
var cat = Cat();

speak(pet);
speak(dog);
speak(cat);
```

Based on the type of each variable another path in the program is chosen. This is how branching can be achieved using dynamic dispatch.

### Challenge 9.2

Instead of looping *recursive* function calls can be used to execute code several times. Functional programming languages like *Haskell* or *Scheme* use recursion instead of loops because they cannot store and modify the state of a variable.

### Challenge 9.3

Parsing the new keyword into an AST node:

```
private Stmt breakStatement()
{
    Token position = previous();
    consume(SEMICOLON, "Expected ';' after 'break'.");
    return new Stmt.Break(position);
}
```

Throw a custom exception when interpreting a break statement so that the outer loop know to stop execution:

```
@Override
public Void visitBreakStmt(Break statement)
{
    throw new BreakException();
}
```

Change the loop interpretation accordingly:

```
@Override
public Void visitWhileStmt(While statement)
{
    while (isTruthy(evaluate(statement.condition)))
    {
        try
        {
            execute(statement.body);
        }
        catch (Interpreter.BreakException breakLoop)
        {
            break;
        }
    }
    return null;
}
```

# Chapter 10: Functions

## Challenge 10.1

Guess: Maybe because Smalltalk is compiled and not interpreted so we do not check for the number of arguments at runtime but at compile time.

Wikipedia: "Smalltalk programs are usually compiled to bytecode, which is then interpreted by a virtual machine or dynamically translated into machine-native code."

## Challenge 10.2

Change grammar to:

```
funDecl -> "fun" function ;
function -> IDENTIFIER? "(" parameters? ")" block ;
```

That means a function name is optional. If the name is omitted we have a *lamda* or *anonymous* function at hand. This approach does not seem to work because lambdas are not statements but expressions which immediately return the function.

Add new expression for this purpose:

```
static class Lambda extends Expr
{
    Lambda(Stmt.Function functionStmt)
    {
        this.functionStmt = functionStmt;
    }

    @Override
    <R> R accept(Visitor<R> visitor)
    {
        return visitor.visitLambdaExpr(this);
    }

    final Stmt.Function functionStmt;
}
```

Parse a lambda expression as a primary expression if we match *FUN*. Parse code:

```
private Expr.Lambda lambda()
{
    // For this type we do not consume an *IDENTIFIER* for the name of the
function.
    Stmt.Function functionStmt = function("anonymous function");
```

```
        return new Expr.Lambda(functionStmt);
    }
```

Interpreter code:

```
@Override
public Object visitLambdaExpr(Lambda expr)
{
    return new LoxFunction(expr.functionStmt, environment);
}
```

### Challenge 10.3

My expectation is that such a program should print "5". The parameters are in another scope compared to the variables declared in the body because while parsing the body we assign a new environment to it.

```
fun scope(a)
{
    var a = "local";
    print a;
}

scope(5);
```

The expectation is wrong. The program prints "local". This is because of the reason already explained above but the conclusion I made was wrong. If we assign a new environment to the body the variable "a" is declared in this environment and therefore if we print "a" we search for its declaration in the inner most environment first which is of course the environment of the function body so "local" is printed.

JavaScript and Python as far as I can tell behave in the same way.

## Chapter 11: Resolving and Binding