Task 1

a) match Java format strings

The regex is located in `java.util.Formatter` source code https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/Formatter.java. The variable is called `FORMAT_SPECIFIER`.

```
"%(\\d+\\$)?([-#+ 0,(\\<]*)?(\\d+)?(\\.\\d+)?([tT])?([a-zA-Z%])"
```

To get the output, I will collect all match begin and end positions in a Queue data structure. Then pass the queue and the entire text into a function. Each iteration will get the head of the queue. The first `if` is for the case when there is text after the last match left. The second `if` exists so that it will not add TEXT when FORMAT is the first part of the string or when two of them are next to each other.

```
public static void print(Queue<Format> lst, String text) {
  var strBuilder = new StringBuilder();
  var index = 0;
  while (index < text.length()) {
    var format = lst.poll();
    if (format == null) {
      strBuilder.append(String.format("TEXT(%s)",
          text.substring(index, text.length())));
      break;
    }
    if (format.begin != 0 && format.begin != index) {
      strBuilder.append(String.format(
          "TEXT(%s)", text.substring(index, format.begin)));
    }
    strBuilder.append(String.format("FORMAT(%s)",
        text.substring(format.begin, format.end)));
    index = format.end;
  }
  System.out.println(strBuilder);
}
```

b) writing ANTLR4 lexer rules for 12-hour clock

From reading the Wikipedia entry https://en.wikipedia.org/wiki/12-hour_clock. I came up with following lexer rules:

```
lexer grammar Aufgabe2Lexer;

Clock: WORD
     | TIME
     ;

WORD: 'Midnight'
    | 'Noon'
    | '12' WS 'midnight'
    | '12' WS 'noon'
    ;

TIME: HOUR SEPARATOR MINUTE WS PERIOD;

PERIOD: 'a.m.'
      | 'p.m.'
      ;
```

```
SEPARATOR: ':';

HOUR: [1-9]
    | '1'[0-2]
    ;

MINUTE: [0-9]
      | [0-5][0-9]
      ;

WS: [ \t\r\n]+ -> channel(HIDDEN);
```

There is a distinction between using midnight, noon to describe time and using numbers and a period.

Task 2

a) Little language

I came up with a grammar for function calls in the form of `(fun arg1 arg2)`. The first element in the list **MUST** be a symbol.

```
lexer grammar SExpressionLexer;
SYMBOL:  (~([ \t\r\n] | '(' | ')' | '{' | '}' | '[' | ']'))+;

LEFT_PAREN: '(';
RIGHT_PAREN: ')';

LEFT_CURLY: '{';
RIGHT_CURLY: '}';

LEFT_BRACKET: '[';
RIGHT_BRACKET: ']';

WS: [ \t\r\n]+ -> channel(HIDDEN);
```

```
parser grammar SExpressionParser;

options { tokenVocab=SExpressionLexer; }

sexpression: LEFT_PAREN head rest* RIGHT_PAREN
           | LEFT_BRACKET head rest* RIGHT_BRACKET
           | LEFT_CURLY head rest* RIGHT_CURLY
           ;

head: SYMBOL;

rest: SYMBOL
    | sexpression
    ;
```

It does not matter which `LEFT` `RIGHT` pair is used, they only need to match each other, which is valid in many Scheme implementations.

Test: `(+ (+ 2 {+ 2 3}) { *  3    5   })`

I also maintain a grammar for Blueprint (https://jwestman.pages.gitlab.gnome.org/blueprint-compiler/) using tree-sitter on https://github.com/huanie/tree-sitter-blueprint :).

AST

My abstract syntax tree will consist of nodes which are sexpressions or literals. A sexpression contains the operation and arguments which are nodes.

```
public interface Node {
    record SExpression(String operation, Iterable<Node> arguments)
            implements Node {
    static SExpression parse(
            SExpressionParser.SexpressionContext sexpression) {
        var arguments = sexpression.rest();
        var head = sexpression.head().getText();
        return new SExpression(head, recurse(arguments));
    }
    private SExpression(String text,
                        List<SExpressionParser.RestContext> rest) {
        this(text, Node.recurse(rest));
    }
    record Literal(String literal) implements Node {
        @Override
        public String toString() {
            return literal;
        }
    }
}
```

Although parsing is usually done with a visitor pattern in OOP (ANTLR4 also prefers it), I use recursion since it feels more natural to me and it is fine with such a small language.

```
private static Iterable<Node> recurse(
        List<SExpressionParser.RestContext> arguments) {
    var argumentAccum = new ArrayList<Node>(arguments.size());
    for (var arg : arguments) {
        var literal = arg.SYMBOL();
        var reduce = arg.sexpression();
        if (literal != null) {
            argumentAccum.add(new Literal(literal.getText()));
        } else if (reduce != null) {
            argumentAccum.add(new SExpression(reduce.head().getText(),
                    reduce.rest()));
        } else {
            throw new RuntimeException(
                    String.format("What is this: %s ?%n", arg.getText()));
        }
    }
    return argumentAccum;
}
```

Using the AST I made a calculator. The accumulator needs to be initialized with the first item in the argument list. Java's `Function<T,R>` only supports functions with one parameter, as a workaround I made use of currying. I also made good use of pattern matching which was introduced in Java 21 which eliminates the visitor pattern in my opinion.

```
public static double calculate(Node.SExpression sexp) {
    var iterator = sexp.arguments().iterator();
    double accum = switch (iterator.next()) {
        case Node.SExpression x -> reduceSexp(x);
        case Node.Literal x -> Double.parseDouble(x.literal());
        default -> throw new IllegalStateException(
```

```
                "Unexpected value: " + sexp.arguments().iterator().next());
    };
    Function<Double, Function<Double, Double>> fun =
            switch (operators.get(sexp.operation())) {
                case Plus -> x -> y -> x + y;
                case Minus -> x -> y -> x - y;
                case Times -> x -> y -> x * y;
                case Divide -> x -> y -> x / y;
            };
    while (iterator.hasNext()) {
        var arg = iterator.next();
        accum = fun.apply(accum).apply(switch (arg) {
            case Node.Literal x -> Double.parseDouble(x.literal());
            case Node.SExpression reduce -> calculate(reduce);
            default -> throw new IllegalStateException(
                    "Unexpected value: " + arg);
        });
    }
    return accum;
}
```