1. Task 1

1.1. 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);
}
```

1.2. 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.

2. Task 2

2.1. 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}) { *  [/ 4 2]    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 :).

2.2. b) AST

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

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.

Iterate through all arguments and check if they are a literal or a sexpression. A literal will be simply be appended to the argument list, a sexpression will recurse (see SExpression record class) before getting appended.

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

Using the AST I made a calculator. I made good use of pattern matching which was introduced in Java 21 which eliminates the visitor pattern in my opinion.

The accumulator needs to be initialized with the first item in the argument list. Then the operation is checked to get the correct function. The subsequent arguments will be passed to the math function or the recursion will continue when encountering a sexpression.

```
private static final Map<String, BiFunction<Double, Double, Double>>
        operators;
static {
    Map<String, BiFunction<Double, Double, Double>> map =
            new HashMap<>();
    map.put("+", (x, y) -> x + y);
    map.put("-", (x, y) -> x - y);
    map.put("*", (x, y) -> x * y);
    map.put("/", (x, y) -> x / y);
    operators = Collections.unmodifiableMap(map);
}

private static double reduceSexp(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());
    };
    var fun = operators.get(sexp.operation());
    while (iterator.hasNext()) {
        var arg = iterator.next();
        accum = fun.apply(accum, switch (arg) {
            case Node.Literal x -> Double.parseDouble(x.literal());
            case Node.SExpression reduce -> reduceSexp(reduce);
            default -> throw new IllegalStateException(
                    "Unexpected value: " + arg);
        });
    }
    return accum;
}
```

3. Task 3

3.1. a) static semantic

There is no static semantic with the current language. I will make it more specialized than `(fun arg1 arg2)`. It will become a calculator, limiting the functions to +, -, * and /. There need to be at least 2 arguments.

Static semantic is now in the number literals to see if the number literals fit in the number range.

Only a few changes to the lexer and parser were done.

Parser:

```
sexpression: LEFT_PAREN head arg arg+ RIGHT_PAREN
           | LEFT_BRACKET head arg arg+ RIGHT_BRACKET
           | LEFT_CURLY head arg arg+ RIGHT_CURLY
           ;

head: PLUS | MINUS | DIVIDE | TIMES;

arg: Float | Integer | Long | Double
   | sexpression
   ;
```

Lexer:

```
fragment Digits: ([0-9])+ ;

Float: Digits '.' Digits FloatSuffix?;
```

```
Double: Digits '.' Digits DoubleSuffix;

Integer: Digits;

Long: Digits LongSuffix;

fragment LongSuffix: [lL];
fragment DoubleSuffix: [Dd];
fragment FloatSuffix: [fF];
```

The only significant change in the code towards building the AST is:

```
public Literal(TerminalNode terminal) {
    Function<String,
            Map.Entry<NumberType, Function<String, Number>>> fun =
            x -> switch (terminal.getSymbol().getType()) {
                case SExpressionLexer.Integer ->
                        Map.entry(NumberType.Integer,
                                o -> Integer.parseInt(o));
                case SExpressionLexer.Double ->
                        Map.entry(NumberType.Double,
                                o -> Double.parseDouble(o));
                case SExpressionLexer.Long ->
                        Map.entry(NumberType.Long,
                                o -> Long.parseLong(o));
                case SExpressionLexer.Float ->
                        Map.entry(NumberType.Float,
                                o -> Float.parseFloat(o));
                default -> throw new RuntimeException(
                        "What is this: " + x);
            };
    this.literal = parseValue(terminal, fun);
}

private Number parseValue(TerminalNode string,
                          Function<String, Map.Entry<NumberType, Function<String,
Number>>> parseFun) {
    var noSuffix = removeSuffix(string.getText());
    var f = parseFun.apply(noSuffix);
    try {
        var number = f.getValue().apply(noSuffix);
        if (List.of(new Number[]{Double.POSITIVE_INFINITY,
                Double.NEGATIVE_INFINITY, Double.NaN,
                Float.POSITIVE_INFINITY,
                Float.NEGATIVE_INFINITY,
                Float.NaN}).contains(number)) {
            throw new NumberFormatException();
        } else {
            return number;
        }
    } catch (NumberFormatException e) {
        Node.errors.add(String.format("%s is not a %s: %s:%s",
                string.getText(),
                f.getKey(),
                string.getSymbol().getLine(),
                string.getSymbol().getCharPositionInLine()));
        return Float.NaN;
    }
}
```

This will collect errors if Java couldn't parse the number to the format that was parsed. At the end of building the AST, it will report all the errors (not shown here).

`(+ (+ 2  {+ 2.23f 2323.23d 3.23}) { *  [/ 4L 2] 223427836467283467823478623487623 4   }`), will report that `2234278364672834678234786234876234` is not an Integer.

The calculator from last task (Section 2.2), remains unchanged.

3.2. b) Dynamic semantic

See Section 2.2. It is an interpreter, calculating arithmetic expressions.

4. Task 4

4.1. a) Procudural

```
readLines(Files.newBufferedReader(input), lines);
removeEmptyLines(lines);
removeShortLines(lines);
int n = totalLineLengths(lines);
```

These lines in the main method already show a procedural step by step approach. The procedures produce side effects by modifying the passed list.

```
private static void readLines(BufferedReader buffer, LinkedList<String> lines) {
    try {
        String line;
        while ((line = buffer.readLine()) != null) {
            lines.add(line);
        }
    } catch (Exception e) {
        System.out.println("Hi");
    }
}
```

Here a temporary variable `line` is created to read the buffer, eventually emptying it. The `line` is added to the passed `lines` list, which is an output parameter.

```
private static void removeEmptyLines(LinkedList<String> lines) {
    var collect = new LinkedList<String>();
    for (var i = 0; i < lines.size(); i++) {
        var line = lines.get(i);
        if (!line.isBlank()) {
            collect.add(line);
        }
    }
    lines.clear();
    lines.addAll(collect);
}
```

This procedure is tricky when using an imperative style because some people might want to directly modify the `lines` list. Modifying the list is incorrect as it will invalidate the loop through the elements. First up a `collect` list is created to hold the results. Looping through the elements of the `lines` list to only add lines that are not blank. At last the list with the results is replaced with the original list (which might not be as fast).

```
private static void removeShortLines(LinkedList<String> lines) {
    var collect = new LinkedList<String>();
    for (var i = 0; i < lines.size(); i++) {
        var line = lines.get(i);
        if (MIN_LENGTH <= line.length()) {
            collect.add(line);
        }
    }
    lines.clear();
    lines.addAll(collect);
}
```

Here the same approach was taken.

```
private static int totalLineLengths(LinkedList<String> lines) {
    var count = 0;
    for (var i = 0; i < lines.size(); i++) {
        count += lines.get(i).length();
    }
    return count;
}
```

Iterating over the list to count the length of all lines which are added to a `count` counter.

This style usually requires more typing, uses local variables to accumulate results and is heavy on side effects.

4.2. b) Functional

```
var n = Files.newBufferedReader(input).lines().filter(l -> !l.isBlank() && MIN_LENGTH
<= l.length()).mapToInt(l -> l.length()).sum();
```

A one liner which explains what kind of transformations to the stream should be done with lambdas.

4.3. c)

- Procedural: `result = 14242 (5005 microsec)`
- Functional: `result = 14242 (5352 microsec)`

The procedural version is still faster despite the inefficient filter procedures in Java.

5. Task 5

5.1. Matching table

```
?- [X,Y,Z] = [john,likes,fish].
X = john,
Y= likes,
Z = fish

?- [cat] = [X|Y].
X = cat,
Y = [] % The end of a list is the empty list (cons 1 (cons 2 '())) '(1 2)

?- [X,Y|Z] = [mary,likes,wine].
X = mary,
Y = likes,
Z = [wine] % the rest starting from the 2nd element, same reason as above

?- [[the,Y]|Z] = [[X,hare],[is,here]].
X= the,
Y = hare,
Z = [[is,here]] % the rest starting from the 1st element, the rest is a list!

?- [golden|T] = [golden,norfolk].
T = [norfolk] % the rest starting from the 1st element

?- [white,horse] = [horse,X].
false % not possible

?- [white|Q] = [P,horse].
P = white, % the first element from left
Q = [horse] % the rest starting from first element
```

5.2. Factorial

```
factorial(N,R) :- fakAcc(N,1,R).
facAcc(0,R,R).
facAcc(N,Acc,R) :-
    Acc1 is Acc*N,
    N1 is N-1,
    facAcc(N1,Acc1,R), !.
```

Accumulator solution. The base case where unifying R with R is necessary to transfer the Result to R. `Acc1 is Acc*N` computes the factorial, `N1 is N-1` is the iteration step. `, !` at the end is optional but makes the output remove the `false`.

5.3. Call history

```
append([],L,L).
append([H|T1],L,[H|T2]) :- append(T1,L,T2).
```

5.3.1. append(X,Y,[1,2,3,4]).

```
append(X,Y,[1,2,3,4]).
% only base case
X = [],
Y = [1,2,3,4],
L = [1,2,3,4]
% first recurse and then base case
append([1|T1],Y,[1|[2,3,4]]) :- append(T1,Y,[2,3,4]).
append(T1,Y,[2,3,4])
T1 = []
Y = [2,3,4]
L = [2,3,4]
X = [1] % backtrack from recursion T1 = [], [1|[]] = [1]
% 2 recusion and then base
T2 = []
T1 = [2]
Y = [3,4]
X = [1,2] % [1|T1] but [T1|T2]
% full recursion
X = [1,2,3,4]
```

5.3.2. append(X,[1,2,3,4],Y).

```
append(X,[1,2,3,4],Y).
% only base case
X = [],
Y = [1,2,3,4]
% 1 recursion
append([H|T1],[1,2,3,4],[H|T2]) :- append(T1,[1,2,3,4],T2).
T1 = [],
T2 = [1,2,3,4],
X = [H],
Y = [H,1,2,3,4] % H could not be found
% 2 recursions
append(H|T1,[1,2,3,4],[H|T2]) :- append(T1,[1,2,3,4],T2).
append(H1|T3,[1,2,3,4],[H1|T4]) :- append(T3,[1,2,3,4],T4).
T3 = [],
T4 = [1,2,3,4],
[H|T1] = [H1|T3]
X = [H|H1],
T2 = [H1|[1,2,3,4]],
Y = [H,H1,1,2,3,4],
% the recursion can go forever
```

5.4. sum

```
sum(L,R) :- sumAcc(L,0,R).
sumAcc([],R,R).
sumAcc([H|T],Acc,R) :-
    Acc1 is H+Acc,
    sumAcc(T,Acc1,R).
```

Another accumulator solution. Accumulator is initialized with 0 and the head of the list is added to it in every iteration until the list is empty.

5.5. Train connections

```
zug(konstanz, 08.39, offenburg, 10.59).
zug(konstanz, 08.39, karlsruhe, 11.49).
zug(konstanz, 08.53, singen, 09.26).
zug(singen, 09.37, stuttgart, 11.32).
zug(offenburg, 11.27, mannheim, 12.24).
zug(karlsruhe, 12.06, mainz, 13.47).
zug(stuttgart, 11.51, mannheim, 12.28).
zug(mannheim, 12.39, mainz, 13.18).

verbindung(Start,At,End,Plan) :- verbindungAcc(Start,At,End,[],Plan).
verbindungAcc(End,_,End,Plan,Plan).
verbindungAcc(Start,At,End,Acc,Plan) :-
    zug(Start,Leave,Stop,Arrive),
    Leave > At,
    append(Acc,[zug(Start,Leave,Stop,Arrive)],X),
    verbindungAcc(Stop,Arrive,End,X,Plan).
```

To get a train plan, my approach is to check from the starting station the trains that come to that station and this will repeat until the train reaches the end station.

`zug(Start,Leave,Stop,Arrive)` will find the station that have a connection from `Start`. The other variables will be filled out. This solution is only valid if the `Arrival` of the train is after the time `At` you want to start at. When it is valid I collect this connection in the accumulator. This will repeat until the end station is reached. It will give all possible connections.

6. Task 6

The template is written as follows:

`delimiters "$", "$"`, will replace going to “expression mode” `<expr>` with `$expr$`.

The start of the template:

```
class_table(n) ::= <<
<!DOCTYPE html>
<html lang="de">
<head>
<style type="text/css">
th, td { border-bottom: thin solid; padding: 4px; text-align: left; }
td { font-family: monospace }
</style>
</head>
<body>
<h1>Sprachkonzepte, Aufgabe 6</h1>
$n:inspect(); separator="\n"$
</body>
</html>
>>
```

It is the HTML document. `$n:inspect(); separator="\n"$` will call the `inspect` template with every element in `n` and inserts a newline after each call.

```
inspect(o) ::= <<
<h2>class $o.name$:</h2>
<table>
<tbody>
$if(o.interface)$
<tr><th>Methods</th></tr>
<tr><td>
$o.methods; separator="<br>\n"$
</td></tr>
$else$
<tr><th>Interface</th><th>Methods</th></tr>
$o.interfaces:interface(); separator="\n"$
$endif$
</tbody>
</table>
<br>
>>
```

`$o.name$` will be the fully qualified name. Furthermore a distinction between interface and class must be made. For interfaces only a box with the methods `$o.methods; separator="<br>\n"$` is created. Classes will call another template for each interface they implement.

```
interface(i) ::= <<
<tr>
<td valign="top">$i.name$</td>
<td>
$i.methods; separator="<br>\n"$
</td>
</tr>
>>
```

Each table row consists of a cell with the fully qualified name `$i.name$` and another cell with their methods `$i.methods; separator="<br>\n"$`.

7. Task 7

The script language I use is Emacs Lisp. Following code is a contribution to a major-mode (provides language relevant editing, syntax-highlighting, navigation) https://git.sr.ht/~meow_king/typst-ts-mode.

Diff: `git diff 361dfe60:typst-ts-mode.el aad1cde1:typst-ts-mode.el`

```elisp
(require 'treesit)
(require 'compile)
(require 'outline)

(defconst typst-ts-mode-outline-regexp "=+ "
  "Regexp identifying Typst header.")

(defun typst-ts-mode-outline-level ()
  "Return the level of the heading at point."
  (save-excursion
    (end-of-line)
    (if (re-search-backward "^=+ " nil t)
  (1- (- (match-end 0) (match-beginning 0)))
      0)))

(defconst typst-ts-mode-outline-heading-alist
  '(("= " . 1)
    ("== " . 2)
    ("=== " . 3)
    ("==== " . 4)
    ("===== " . 5)
    ("====== " . 6)))

(defun typst-ts-mode-heading--at-point-p ()
  "Whether the current line is a heading.
Return the heading node when yes otherwise nil."
  (let ((node (treesit-node-parent
         (treesit-node-at
                (save-excursion
                  (beginning-of-line-text)
                  (point))))))
    (if (string= (treesit-node-type node) "heading")
  node
      nil)))

(defun typst-ts-mode-heading-up ()
  "Switch the current heading with the heading above."
  (interactive)
  (typst-ts-mode-meta--dwim 'up))

(defun typst-ts-mode-heading-down ()
  "Switch the current heading with the heading below."
  (interactive)
  (typst-ts-mode-meta--dwim 'down))

(defun typst-ts-mode-heading-increase ()
  "Increase the heading level."
  (interactive)
  (typst-ts-mode-meta--dwim 'right))

(defun typst-ts-mode-heading-decrease ()
  "Decrease heading level."
  (interactive)
  (typst-ts-mode-meta--dwim 'left))

(defun typst-ts-mode-meta--dwim (direction)
  "Do something depending on the context with meta key + DIRECTION.
`left': `typst-ts-mode-heading-decrease',
`right': `typst-ts-mode-heading-increase',
`up': `typst-ts-mode-heading-up',
`down': `typst-ts-mode-heading-down'.
When there is no relevant action to do it will execute the relevant function in
the `GLOBAL-MAP' (example: `right-word')."
  (let ((heading (typst-ts-mode-heading--at-point-p))
  ;; car function, cdr string of function for `substitute-command-keys'
  (call-me/string
   (pcase direction
     ('left
      (cons #'outline-promote
      "\\[typst-ts-mode-heading-decrease]"))
     ('right
      (cons #'outline-demote
      "\\[typst-ts-mode-heading-decrease]"))
     ('up
      (cons #'outline-move-subtree-up
      "\\[typst-ts-mode-heading-up]"))
     ('down
      (cons #'outline-move-subtree-down
      "\\[typst-ts-mode-heading-down]"))
     (_ (error "%s is not one of: `right' `left'" direction)))))
    (if heading
  (call-interactively (car call-me/string))
      (call-interactively
       (keymap-lookup global-map (substitute-command-keys (cdr call-me/string)))))))

(defvar typst-ts-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "C-c C-c c") #'typst-ts-mode-compile-and-preview)
    (define-key map (kbd "C-c C-c C") #'typst-ts-mode-compile)
    (define-key map (kbd "C-c C-c w") #'typst-ts-mode-watch-toggle)
    (define-key map (kbd "C-c C-c p") #'typst-ts-mode-preview)
    (define-key map (kbd "M-<left>") #'typst-ts-mode-heading-decrease)
    (define-key map (kbd "M-<right>") #'typst-ts-mode-heading-increase)
    (define-key map (kbd "M-<down>") #'typst-ts-mode-heading-down)
    (define-key map (kbd "M-<up>") #'typst-ts-mode-heading-up)
    (define-key map (kbd "TAB") #'typst-ts-mode-cycle)
    map))
```

The functionality that was added is manipulating and navigating headings in a markup language called “Typst”. Switching headings of the same level up or down. Decreasing or increasing the level of the heading and its children. Most of the functionality is provided by the `outline` library (after I discovered that it exists). I only added glue code to either do something with the heading when the cursor is pointing at one or execute the default command for given keybinding when it is not pointing at one.

Emacs Lisp is a multi-paradigm (more imperative leaning) high level scripting language.

It is garbage collected, so I do not need to care about memory management.

It has dynamic typing which is showcased in `typst-ts-mode-heading--at-point-p` returning either a nil or an object. Anything that is not nil or empty list is true. Exception handling is not enforced, I do not know what can be thrown and neither do I handle any.

I extensively used the interactive nature of the language. Any expression can be tested in a REPL or by typing it and then pressing `Ctrl+x Ctrl+e` after it. I used it to quickly explore the effect of functions I do not know about and to fix my own code.

Many functions are implemented in C for speed. Taking `save-excursion` as an example, it will keep the cursor position for the user even when the functions in the body move the pointer.

I don’t use dynamic scoping and had to do workarounds to avoid it (nobody wants it). A comment on the first line of the file is needed to enable lexical scoping. Even with the option enabled, only variables that were declared in `let` are lexically scoped.