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 `223427836467283467823478623487623` 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. Prolog

4.1. 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
```

4.2. 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`.

4.3. Call history

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

4.3.1. 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]
```

4.3.2. 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
```

4.4. 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.

4.5. 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.