Note: The order of tasks was fixed and the same for all participants without the possibility of returning to previous task. However, possible answers for the task were randomly assigned between participants.

1) There is no context-free grammar for the language $a^n b^n c^n$, so we transfer the problem of the same number of occurrences of a, b and c to the semantics. In this case, the context-free grammar is actually for the language $a^i b^j c^k$, and we check in the semantics whether i=j=k holds. A couple of examples of programs and their meanings:

`aaabbbccc` -> ok=true

`aabbccc` -> ok=false

`abbcc` -> ok=false

Which attribute grammar correctly describes the semantics of the programming language $a^n b^n c^n$?

a)

```
language AnBnCn {
  lexicon {
    TokenA a
    TokenB b
    TokenC c
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.val; boolean *.ok;
  rule S {
    S ::= A B C compute {
      S.ok = (A.val == B.val) && (B.val == C.val);
    };
  }
  rule A {
    A ::= a A compute {
      A[0].val = 1 + A[1].val;
    };
    A ::= a compute {
      A.val = 1;
    };
  }
  rule B {
    B ::= b B compute {
      B[0].val = 1 + B[1].val;
    };
    B ::= b compute {
      B.val = 1;
    };
  }
  rule C {
    C ::= c C compute {
      C[0].val = 1 + C[1].val;
    };
```

```
    C ::= c compute {
      C.val = 1;
    };
  }
}
```

b)

```
language AnBnCn {
  lexicon {
    TokenA a
    TokenB b
    TokenC c
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.val; boolean *.ok;
  rule S {
    S ::= A B C compute {
      S.ok = (A.val == B.val) && (B.val == C.val);
    };
  }
  rule A {
    A ::= a A compute {
      A[0].val = A[1].val;
    };
    A ::= a compute {
      A.val = 1;
    };
  }
  rule B {
    B ::= b B compute {
      B[0].val = B[1].val;
    };
    B ::= b compute {
      B.val = 1;
    };
  }
  rule C {
    C ::= c C compute {
      C[0].val = C[1].val;
    };
    C ::= c compute {
      C.val = 1;
    };
  }
}
```

c)

```
language AnBnCn {
  lexicon {
    TokenA a
    TokenB b
    TokenC c
    ignore [\ \0x0D\0x0A\0x09]+

  }
  attributes int *.val;
             boolean *.ok;

  rule S {
    S ::= A B C   compute {
      S.ok = (A.val == B.val) && (B.val == C.val);
    };
  }
  rule A {
    A ::= a A compute {
     A[0].val = (1+1)+(A[1].val+1);
    };
    A ::= a compute {
      A.val = (1+1)+1;
    };
  }
  rule B {
    B ::= b B compute {
     B[0].val = 1+B[1].val;
    };
    B ::= b compute {
      B.val = (1+1)+1;
    };
  }
  rule C {
    C ::= c C compute {
     C[0].val = C[1].val;
    };
    C ::= c compute {
      C.val = 1+1;
    };
  }
}
```

d)

```
language AnBnCn {
  lexicon {
    TokenA a
    TokenB b
    TokenC c
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.val; boolean *.ok;
  rule S {
    S ::= A B C compute {
      S.ok = A.val==(B.val+C.val);
    };
  }
  rule A {
    A ::= a A compute {
     A[0].val = (1+A[1].val);
    };
    A ::= a compute {
      A.val = 1;
    };
  }
  rule B {
    B ::= b B compute {
     B[0].val = B[1].val+1;
    };
    B ::= b compute {
      B.val = 1;
    };
  }
  rule C {
    C ::= c C compute {
     C[0].val = 1+C[1].val;
    };
    C ::= c compute {
      C.val = 1;
    };
  }
}
```

e)

None of the provided answers.

2) The ExprLA language is a simple arithmetic expression language with a + and - operators. The attribute grammar of type L below (containing synthesized and inherited attributes) gives the meaning of an arithmetic expression. A couple of examples of programs and their meanings:

`2 - 4 + 5` -> vals=3

`1 - 2` -> vals=-1

`1 + 2` -> vals=3

Which attribute grammar correctly describes the semantics of the ExprLA programming language?

a)

```
language ExprLA {
  lexicon {
    Int [0-9]+
    Operator \+ | \-
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.vals, *.invali;

  rule exp1 {
   E ::= T EE compute {
    E.vals = EE.vals;
    EE.invali = T.vals;
   };
  }
  rule exp2 {
   EE ::= + T EE compute {
    EE[0].vals = EE[1].vals;
    EE[1].invali = EE[0].invali+T.vals;
   };
  }
  rule exp3 {
   EE ::= - T EE compute {
    EE[0].vals = EE[1].vals;
    EE[1].invali = EE[0].invali-T.vals;
   };
  }
  rule exp4 {
   EE ::= epsilon compute {
    EE.vals = EE.invali;
   };
  }
  rule exp5 {
   T ::= #Int compute {
    T.vals = Integer.parseInt(#Int.value());
   };
  }
}
```

b)

```
language ExprLA {
  lexicon {
    Int [0-9]+
    Operator \+ | \-
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.vals, *.invali;

  rule exp1 {
   E ::= T EE compute {
    E.vals = EE.vals;
    EE.invali = T.vals;
   };
  }
  rule exp2 {
   EE ::= + T EE compute {
    EE[0].vals = EE[1].vals;
    EE[1].invali = EE[0].invali+T.vals;
   };
  }
  rule exp3 {
   EE ::= - T EE compute {
    EE[0].vals = EE[1].vals;
    EE[1].invali = EE[0].invali-T.vals;
   };
  }
  rule exp4 {
   EE ::= epsilon compute {
    EE.vals = 0;
   };
  }
  rule exp5 {
   T ::= #Int compute {
    T.vals = Integer.parseInt(#Int.value());
   };
  }
}
```

c)

```
language ExprLA {
  lexicon {
    Int [0-9]+
    Operator \+ | \-
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.vals, *.invali;

 rule exp1 {
   E ::= T EE compute {
    E.vals = EE.vals+EE.invali;
    EE.invali = T.vals;
   };
```

```
 }
 rule exp2 {
   EE ::= + T EE compute {
    EE[0].vals = T.vals;
    EE[1].invali = EE[0].invali+EE[0].invali;
   };
 }
 rule exp3 {
   EE ::= - T EE compute {
    EE[0].vals = EE[1].vals-T.vals;
    EE[1].invali = EE[0].invali;
   };
 }
 rule exp4 {
   EE ::= epsilon compute {
    EE.vals = EE.invali;
   };
 }
 rule exp5 {
   T ::= #Int compute {
    T.vals = Integer.parseInt(#Int.value());
   };
 }
}
```

d)

```
language ExprLA {
  lexicon {
    Int [0-9]+
    Operator \+ | \-
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.vals, *.invali;

  rule exp1 {
   E ::= T EE compute {
    E.vals = T.vals-EE.vals;
    EE.invali = T.vals;
   };
  }
  rule exp2 {
   EE ::= + T EE compute {
    EE[0].vals = EE[1].vals-T.vals;
    EE[1].invali = T.vals;
   };
  }
  rule exp3 {
   EE ::= - T EE compute {
    EE[0].vals = EE[1].invali+EE[1].vals;
    EE[1].invali = T.vals;
   };
  }
```

```
  rule exp4 {
   EE ::= epsilon compute {
    EE.vals = EE.invali;
   };
  }
  rule exp5 {
   T ::= #Int compute {
    T.vals = Integer.parseInt(#Int.value());
   };
  }
}
```

e)

None of the provided answers.

3. The Robot language is a simple example of a domain-specific language. The meaning of Robot language programs is defined as the final position of the robot after the execution of the program. At the beginning, the robot is located at position (0,0). We control the robot in 4 directions (up, down, left, right), whereby with each command (up, down, left, right) we move the robot by one unit in the corresponding direction. Commands are preceded by the keyword begin and commands are followed by the keyword end. The program begin end is a syntactically and semantically correct program with the meaning (0, 0). A couple of examples of programs and their meanings:

`begin end` -> outx=0, outy=0

`begin left left end` -> outx=-2, outy=0

`begin right up left down up end` -> outx=0, outy=1

Which attribute grammar correctly describes the semantics of the Robot programming language?

a)

```
language Robot {
    lexicon {
        Command      left | right | up | down
        ReservedWord begin | end
        ignore [\0x0D\0x0A\ ]
    }
    attributes int *.inx; int *.iny;
               int *.outx; int *.outy;
    rule start {
       START ::= begin COMMANDS end compute {
         START.outx = COMMANDS.outx;
         START.outy = COMMANDS.outy;
         COMMANDS.inx = 0;
         COMMANDS.iny = 0;
      };
    }
    rule commands {
       COMMANDS ::= COMMAND COMMANDS compute {
```

```
          COMMANDS[0].outx = COMMANDS[1].outx;
          COMMANDS[0].outy = COMMANDS[1].outy;
          COMMAND.inx = COMMANDS[0].inx;
          COMMAND.iny = COMMANDS[0].iny;
          COMMANDS[1].inx = COMMAND.outx;
          COMMANDS[1].iny = COMMAND.outy;
        }
      | epsilon compute {
          COMMANDS[0].outx = COMMANDS[0].inx;
          COMMANDS[0].outy = COMMANDS[0].iny;
        };
    }
    rule command {
        COMMAND ::= left compute {
          COMMAND.outx = COMMAND.inx - 1;
          COMMAND.outy = COMMAND.iny;
        };
        COMMAND ::= right compute {
          COMMAND.outx = COMMAND.inx + 1;
          COMMAND.outy = COMMAND.iny;
        };
        COMMAND ::= up compute {
          COMMAND.outx = COMMAND.inx;
          COMMAND.outy = COMMAND.iny + 1;
        };
        COMMAND ::= down compute {
          COMMAND.outx = COMMAND.inx;
          COMMAND.outy = COMMAND.iny - 1;
        };
   }
}
```

b)

```
language Robot {
    lexicon {
        Command      left | right | up | down
        ReservedWord begin | end
        ignore [\0x0D\0x0A\ ]
    }
    attributes int *.inx; int *.iny;
               int *.outx; int *.outy;
    rule start {
        START ::= begin COMMANDS end compute {
          START.outx = COMMANDS.outx;
          START.outy = COMMANDS.outy;
          COMMANDS.inx = 0;
          COMMANDS.iny = 0;
       };
    }
    rule commands {
        COMMANDS ::= COMMAND COMMANDS compute {
          COMMANDS[0].outx = COMMAND.outx;
```

```
          COMMANDS[0].outy = COMMAND.outy;
          COMMAND.inx = COMMANDS[0].inx;
          COMMAND.iny = COMMANDS[0].iny;
          COMMANDS[1].inx = COMMANDS[0].outx;
          COMMANDS[1].iny = COMMANDS[0].outy;
        }
      | epsilon compute {
          COMMANDS[0].outx = COMMANDS[0].inx;
          COMMANDS[0].outy = COMMANDS[0].iny;
        };
    }
    rule command {
        COMMAND ::= left compute {
          COMMAND.outx = COMMAND.inx - 1;
          COMMAND.outy = COMMAND.iny;
        };
        COMMAND ::= right compute {
          COMMAND.outx = COMMAND.inx + 1;
          COMMAND.outy = COMMAND.iny;
        };
        COMMAND ::= up compute {
          COMMAND.outx = COMMAND.inx;
          COMMAND.outy = COMMAND.iny + 1;
        };
        COMMAND ::= down compute {
          COMMAND.outx = COMMAND.inx;
          COMMAND.outy = COMMAND.iny - 1;
        };
   }
}
```

c)

```
language Robot {
    lexicon {
        Command      left | right | up | down
        ReservedWord begin | end
        ignore [\0x0D\0x0A\ ]
    }
    attributes int *.inx; int *.iny;
               int *.outx; int *.outy;
    rule start {
        START ::= begin COMMANDS end compute {
          START.outx = COMMANDS.outx;
          START.outy = COMMANDS.outy;
          COMMANDS.inx = 0;
          COMMANDS.iny = 0;
       };
    }
    rule commands {
        COMMANDS ::= COMMAND COMMANDS compute {
          COMMANDS[0].outx = COMMANDS[1].outx;
          COMMANDS[0].outy = COMMANDS[1].outy;
```

```
          COMMAND.inx = COMMANDS[0].inx;
          COMMAND.iny = COMMANDS[0].iny;
          COMMANDS[1].inx = COMMAND.outx;
          COMMANDS[1].iny = COMMAND.outy;
        }
      | epsilon compute {
          COMMANDS[0].outx = 0;
          COMMANDS[0].outy = 0;
        };
    }
    rule command {
        COMMAND ::= left compute {
          COMMAND.outx = COMMAND.inx - 1;
          COMMAND.outy = COMMAND.iny;
        };
        COMMAND ::= right compute {
          COMMAND.outx = COMMAND.inx + 1;
          COMMAND.outy = COMMAND.iny;
        };
        COMMAND ::= up compute {
          COMMAND.outx = COMMAND.inx;
          COMMAND.outy = COMMAND.iny + 1;
        };
        COMMAND ::= down compute {
          COMMAND.outx = COMMAND.inx;
          COMMAND.outy = COMMAND.iny - 1;
        };
   }
}
```

d)

```
language Robot {
    lexicon {
        Command      left | right | up | down
        ReservedWord begin | end
        ignore [\0x0D\0x0A\ ]
    }
    attributes int *.inx; int *.iny;
               int *.outx; int *.outy;

    rule start {
        START ::= begin COMMANDS end compute {
          START.outx = COMMANDS.outy;
          START.outy = COMMANDS.outx;
          COMMANDS.inx = 0;
          COMMANDS.iny = 0;
        };
      }
    rule commands {
        COMMANDS ::= COMMAND COMMANDS compute {
          COMMANDS[0].outx = COMMANDS[1].outx+COMMAND.outy;
          COMMANDS[0].outy = COMMANDS[1].outy+COMMAND.outx;
```

```
            COMMAND.inx = COMMANDS[0].iny;
            COMMAND.iny = COMMANDS[0].inx;
            COMMANDS[1].inx = COMMANDS[0].iny;
            COMMANDS[1].iny = 0;
          }
          | epsilon compute {
              COMMANDS.outx = COMMANDS.iny;
              COMMANDS.outy = COMMANDS.inx+1;
          };
      }
      rule command {
          COMMAND ::= left compute {
            COMMAND.outx = 0-1;
            COMMAND.outy = COMMAND.iny;
          };
          COMMAND ::= right compute {
            COMMAND.outx = 1-COMMAND.outy;
            COMMAND.outy = COMMAND.inx;
          };
          COMMAND ::= up compute {
            COMMAND.outx = COMMAND.inx;
            COMMAND.outy = 1;
          };
          COMMAND ::= down compute {
            COMMAND.outx = COMMAND.iny-COMMAND.iny;
            COMMAND.outy = COMMAND.iny-1;
          };
    }
}
```

e)

None of the provided answers.

4) The SimpleWhereExpression language requires an absolutely non-circular AG, since a semantic tree cannot be evaluated with just one walk through the tree. The language contains a simple arithmetic expression with the + operator and contains an integer and a constant that is defined by the arithmetic expression. A couple of examples of programs and their meanings:

`y + 2 where y = 5` -> ok=true, val=7

`y + 2 where z = 5` -> ok=false (since the constant y is not defined), val=7

`z + 10 where z = 0` -> ok=true, val=10

Which attribute grammar correctly describes the semantics of the programming language SimpleWhereExpression?

a)

```
language SimpleWhereExpression {
  lexicon {
    Id [a-z][a-z0-9]*
    Num [0-9]+
    Operator \+ | \=
    keyword where
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.val;
       int *.inval;
       int *.outval;
       String *.konst;
       boolean *.ok;

 rule Prog {
   P ::= E where D compute {
      P.ok = E.ok;
      P.val = E.outval;
      E.konst = D.konst;
      E.inval = D.val;
   };
  }
  rule Exp {
   E ::= #Id + #Num compute {
      E.ok = #Id.value().equals(E.konst);
      E.outval = E.inval+Integer.parseInt(#Num.value());
   };
  }
  rule Dec {
   D ::= #Id = #Num compute {
      D.konst = #Id.value();
      D.val = Integer.parseInt(#Num.value());
   };
  }
}
```

b)

```
language SimpleWhereExpression {
  lexicon {
    Id [a-z][a-z0-9]*
    Num [0-9]+
    Operator \+ | \=
    keyword where
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.val;
       int *.inval;
       int *.outval;
       String *.konst;
       boolean *.ok;
```

```
 rule Prog {
   P ::= E where D compute {
       P.ok = D.konst.equals(E.konst);
       P.val = E.outval;
       E.konst = D.konst;
       E.inval = Integer.parseInt(E.konst);
    };
  }
  rule Exp {
   E ::= #Id + #Num compute {
       E.ok = #Num.value().equals(#Id.value());
       E.outval = Integer.parseInt(#Num.value())+
                  Integer.parseInt(E.konst);
    };
  }
  rule Dec {
   D ::= #Id = #Num compute {
       D.konst = #Num.value();
       D.val = Integer.parseInt(#Id.value());
    };
  }
}
```

c)

```
language SimpleWhereExpression {
  lexicon {
    Id [a-z][a-z0-9]*
    Num [0-9]+
    Operator \+ | \=
    keyword where
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.val;
       int *.inval;
       int *.outval;
       String *.konst;
       boolean *.ok;

 rule Prog {
   P ::= E where D compute {

       P.ok = E.konst.equals(E.konst);
       P.val = E.inval;
       E.konst = D.konst;
       E.inval = E.outval;
    };
  }
  rule Exp {
   E ::= #Id + #Num compute {
       E.ok = E.konst.equals(#Id.value());
       E.outval = Integer.parseInt(#Num.value());
```

```
   };
  }
  rule Dec {
   D ::= #Id = #Num compute {
      D.konst = #Num.value();
      D.val = Integer.parseInt(#Id.value());
   };
  }
}
```

d)

```
language SimpleWhereExpression {
  lexicon {
    Id [a-z][a-z0-9]*
    Num [0-9]+
    Operator \+ | \=
    keyword where
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.val;
       int *.inval;
       int *.outval;
       String *.konst;
       boolean *.ok;

 rule Prog {
   P ::= E where D compute {
      P.ok = E.ok;
      P.val = D.val+E.outval;
      E.konst = D.konst;
      E.inval = D.val;
   };
  }
  rule Exp {
   E ::= #Id + #Num compute {
      E.ok = #Id.value().equals(E.konst);
      E.outval = E.inval+Integer.parseInt(#Num.value());
   };
  }
  rule Dec {
   D ::= #Id = #Num compute {
      D.konst = #Id.value();
      D.val = Integer.parseInt(#Num.value());
   };
  }
}
```

e)

None of the provided answers.

5) Correct the attribute grammar below for the $a^n b^n c^n$ language so that it is correct (incorrect semantic rules are written in red).

```
language AnBnCn {
  lexicon {
    TokenA a
    TokenB b
    TokenC c
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.val; boolean *.ok;
  rule S {
    S ::= A B C compute {
      S.ok = A.val==(B.val+C.val);
    };
  }
  rule A {
    A ::= a A compute {
     A[0].val = (1+A[1].val);
    };
    A ::= a compute {
      A.val = 1;
    };
  }
  rule B {
    B ::= b B compute {
     B[0].val = B[1].val+1;
    };
    B ::= b compute {
      B.val = 1;
    };
  }
  rule C {
    C ::= c C compute {
     C[0].val = 1+C[1].val;
    };
    C ::= c compute {
      C.val = 1;
    };
  }
}
```

6) Correct the attribute grammar below for the ExprLA language so that it is correct (incorrect semantic rules are written in red).

```
language ExprLA {
  lexicon {
    Int [0-9]+
    Operator \+
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes int *.vals, *.invali;
```

```
  rule exp1 {
   E ::= T EE compute {
    E.vals = EE.vals;
    EE.invali = T.vals;
   };
  }
  rule exp2 {
   EE ::= + T EE compute {
    EE[1].invali = EE[0].invali;
    EE[0].vals = EE[1].vals;
   };
  }
  rule exp4 {
   EE ::= epsilon compute {
    EE.vals = EE.invali;
   };
  }
  rule exp5 {
   T ::= #Int compute {
    T.vals = Integer.parseInt(#Int.value());
   };
  }
}
```

7) Correct the below attribute grammar for the Robot language so that it is correct (incorrect semantic rules are written in red).

```
language Robot {
    lexicon {
        Command      left | right | up | down
        ReservedWord begin | end
        ignore [\0x0D\0x0A\ ]
    }
    attributes int *.inx; int *.iny;
               int *.outx; int *.outy;
    rule start {
       START ::= begin COMMANDS end compute {
         START.outx = COMMANDS.outx;
         START.outy = COMMANDS.outy;
         COMMANDS.inx = 0;
         COMMANDS.iny = 0;
      };
    }
    rule commands {
       COMMANDS ::= COMMAND COMMANDS compute {
         COMMAND.iny = COMMANDS[1].iny;
         COMMANDS[0].outx = COMMANDS[1].outx;
         COMMANDS[1].iny = COMMAND.outy;
         COMMANDS[1].inx = COMMAND.outx;
         COMMAND.inx = COMMANDS[0].inx;
         COMMANDS[0].outy = COMMANDS[1].outy;
       }
     | epsilon compute {
```

```
            COMMANDS[0].outx = COMMANDS[0].inx;
            COMMANDS[0].outy = COMMANDS[0].iny;
        };
    }
    rule command {
        COMMAND ::= left compute {
            COMMAND.outx = COMMAND.inx - 1;
            COMMAND.outy = COMMAND.iny;
        };
        COMMAND ::= right compute {
            COMMAND.outx = COMMAND.inx + 1;
            COMMAND.outy = COMMAND.iny;
        };
        COMMAND ::= up compute {
            COMMAND.outx = COMMAND.inx;
            COMMAND.outy = COMMAND.iny + 1;
        };
        COMMAND ::= down compute {
            COMMAND.outx = COMMAND.inx;
            COMMAND.outy = COMMAND.iny - 1;
        };
    }
}
```