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 $a^nb^nc^n$ language. The problem of the same number of occurrences of a and b can be solved with a context-free grammar (a^nb^n language). At the level of semantics, we can check whether the number of occurrences of c is equal to the occurrence of a or b. We actually have the language $a^nb^nc^m$ and at the level of semantics we check if n=m. A couple of examples of programs and their meanings:

`aaabbbccc` -> ok=true

`aabbccc` -> ok=false

`abbcc` -> // syntax error

In the attribute grammar given below, is the val attribute an inherited or a synthesized attribute?

a) synthesized
b) inherited

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

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

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

2) The Assign language contains an assignment command, where we check whether the type of the variable on the left hand side of the assignment command corresponds to the type of the expression on the right hand side of the assignmenmt command (exact match). Here, variable A is of type real, and variables B and C are of type int. A couple of examples of programs and their meanings:

`B = B + C` -> ok=true

`B = A + C` -> ok=false

`A = B + C` -> ok=false

`A = B` -> ok=false

`A = A + A` -> ok=true

In the Attribute Grammar given below, the semantic rule is missing in the first production. Which attribute is not defined?

```
language Assign {
   lexicon {
          Id        A | B | C
          Operator  = | \+
          ignore    [\0x09\0x0A\0x0D\  ]+
   }

   attributes String *.expected_type,  *.actual_type;
              boolean *.ok;


   rule Start {
        ASSIGN ::= VAR = EXPR  compute {
          ASSIGN.ok = EXPR.actual_type.equals(EXPR.expected_type);
        };
     }

   rule Expression {
        EXPR::= VAR + VAR compute {
          EXPR.actual_type =
               (VAR[0].actual_type.equals(VAR[1].actual_type)) ?
                VAR[0].actual_type    : "ERROR";
        }
        |
        VAR compute {
          EXPR.actual_type = VAR.actual_type;
        }
        ;
     }

   rule Variable {
        VAR ::= #Id compute {
             VAR.actual_type = lookup(#Id.value());
        };
     }
   method Environment
     {
        public String lookup(String name)
          {
             if (name.equals("A")) return "real";
             else return "int";
          }
     }
}
```

3) What is the meaning of sentence/program 111 in the following attribute grammar for the language of binary numbers?

```
language XY {
  lexicon {
    Token0 0
    Token1 1
    ignore [\ \0x0D\0x0A\0x09]+
  }
  attributes double *.val, *.pos, *.num;

  rule E {
    E ::= B   compute {
      E.val = B.val; };
  }

  rule B {
    B ::= D BB   compute {
      B.pos = BB.pos+1;
      B.val = BB.val + D.num*java.lang.Math.pow(10.0,B.pos);
    };
  }

  rule BB {
    BB ::= D BB   compute {
      BB[0].pos = BB[1].pos+1;
      BB[0].val = BB[1].val +
                  D.num*java.lang.Math.pow(10.0,BB[0].pos);
    }
    |
    epsilon  compute {
      BB.pos=-1;
      BB.val=0;
    };
  }

  rule D {
    D ::= 0 compute {
     D.num = 0;
    };
    D ::= 1 compute {
      D.num = 1;
    };
  }
}
```

4) Which statement is true for the Attribute Grammar given below:

a) It is circular AG.

b) It is S-attributed grammar.

c) It is L-attributed grammar.

d) It is Absolutely non-circular AG.

e) None of the provided answers.

```
language TEST {
    lexicon {
        keywords  begin | end
           ignore [\0x0D\0x0A\ ]
    }
    attributes int *.inx; int *.outx;

    rule start {
     START ::= begin A end compute {
         START.outx = A.outx;
         A.inx = A.outx;
     };

    }
    rule com {
     A ::= epsilon compute {
         A.outx = A.inx;
     };
    }
 }
```

5) The Desk language contains a simple arithmetic expression followed by constant definitions. The (incomplete) attribute grammar given below converts a Desk language program into simple VM instructions. A couple of examples of programs and their meanings:

`print x + y where x=1, y=2` -> code=LOAD 1 ADD 2 PRINT 0 HALT 0

`print 100 + x where x=1` -> code=LOAD 100 ADD 1 PRINT 0 HALT 0

Do the two rules below (attribute grammars for the Desk language) meet the requirements for L-type attribute grammars? Give an explanation.

```
language Desk {
   lexicon
     {
          Number     [0-9]+
          Id         [a-z]+
          Keyword    print | where
          Operator   = | \+
          Separator  ,
          ignore     [\0x09\0x0A\0x0D\  ]+
     }

   attributes String *.code,  *.name;
```

```
             int *.value;
             Hashtable *.envs, *.envi;
             boolean *.ok;


   rule Start {
        PROGRAM ::= print EXPRESSION CONSTPART  compute {
             PROGRAM.code = CONSTPART.ok ? EXPRESSION.code +
                                 "PRINT 0" + "\n" + "HALT 0" + "\n"
                               : "\n" + "HALT 0" + "\n";
             EXPRESSION.envi = CONSTPART.envs;
          };
     }
   …

   rule Constpart {
        CONSTPART ::= where CONSTDEFLIST compute {
             CONSTPART.ok = CONSTDEFLIST.ok;
             CONSTPART.envs = CONSTDEFLIST.envs;
        }
        | epsilon  compute  {
             CONSTPART.ok = true;
             CONSTPART.envs = new Hashtable();
        };
     }

    …
}
```

6) Which attribute grammar for the $a^n b^n c^n$ language below is incorrect?

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 = 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;
    };
  }
  rule C {
    C ::= c C compute {
     C[0].val = C[1].val;
    };
    C ::= c compute {
      C.val = 1+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 = (B.val*A.val)==C.val;
    };
  }

  rule A {
    A ::= a A compute {
     A[0].val = 3*A[1].val;
    };
    A ::= a compute {
      A.val = (1+1);
    };
  }

  rule B {
    B ::= b B compute {
     B[0].val = (2*B[1].val);
    };
    B ::= b compute {
      B.val = (1+1)+1;
    };
  }

  rule C {
    C ::= c C compute {
     C[0].val = (C[1].val*6);
    };
    C ::= c compute {
      C.val = 1+1+1+1+1+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==2 * C.val) && (B.val==C.val);
    };
  }
  rule A {
    A ::= a A compute {
     A[0].val = 1+(1+A[1].val);
    };
    A ::= a compute {
      A.val = 1+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;
    };
  }
}
```

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) && (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;
    };
  }
}
```

e) They are all correct.

7) The (wrong) attribute grammar below gives the meaning of a simple let expression, where the definition of constants is followed by an arithmetic expression containing constants and integers. A couple of examples of programs and their meanings:

`let y=2, x=3 in y+x+1` -> ok=true, value=6

`let y=2, y=30 in y + 1` -> ok=false (since the constant y is defined twice), value=31

`let y=2 in y + 1` -> ok=true, value=3

`let y=2 in y + x` -> ok=false (since the constant x is not defined), value=2

Which semantic equation is redundant in the first production?

```
language Let {
   lexicon {
          Number     [0-9]+
          Id         [a-z]+
          Keyword    let | in
          Operator   = | \+
          Separator  ,
          ignore     [\0x09\0x0A\0x0D\  ]+
   }

   attributes String *.name;
              int *.value;
              Hashtable *.env;
              boolean *.ok;

   rule Start {
        PROGRAM ::= let CONSTPART in EXPRESSION compute  {
             PROGRAM.value =  EXPRESSION.value;
             EXPRESSION.env = CONSTPART.env;
             EXPRESSION.ok = CONSTPART.ok;
             PROGRAM.ok = EXPRESSION.ok && CONSTPART.ok;
        };
     }

   rule Expression {
        EXPRESSION ::= EXPRESSION + FACTOR compute {
             EXPRESSION[0].value = EXPRESSION[1].value +
                                   FACTOR.value;
             EXPRESSION[1].env = EXPRESSION[0].env;
             FACTOR.env = EXPRESSION[0].env;
             EXPRESSION[0].ok=EXPRESSION[1].ok && FACTOR.ok;
          }
        |  FACTOR compute {
             EXPRESSION[0].value = FACTOR.value;
             FACTOR.env = EXPRESSION[0].env;
             EXPRESSION[0].ok=FACTOR.ok;
          }
          ;
     }

   rule Factor {
        FACTOR ::= CONSTNAME compute {
             FACTOR.ok = FACTOR.env.containsKey(CONSTNAME.name);
             FACTOR.value = FACTOR.ok ?
               Integer.parseInt(FACTOR.env.get(CONSTNAME.name))
               : 0;
            };
        FACTOR ::= #Number compute {
             FACTOR.ok = true;
             FACTOR.value = Integer.parseInt(#Number.value());
          };
   }

   rule Constname {
        CONSTNAME ::= #Id compute {
             CONSTNAME.name = #Id.value();
          };
   }

   rule Constpart {
        CONSTPART ::= CONSTDEFLIST compute {
             CONSTPART.ok = CONSTDEFLIST.ok;
             CONSTPART.env = CONSTDEFLIST.env;
          };
   }

   rule Constdeflist {
        CONSTDEFLIST ::= CONSTDEFLIST , CONSTDEF compute {
             CONSTDEFLIST[0].ok = CONSTDEFLIST[1].ok &&
                  !CONSTDEFLIST[1].env.containsKey(CONSTDEF.name);
             CONSTDEFLIST[0].env = put(CONSTDEFLIST[1].env,
                                     CONSTDEF.name, CONSTDEF.value);
          }
        | CONSTDEF  compute {
             CONSTDEFLIST.ok = true;
             CONSTDEFLIST.env = put(new Hashtable(), CONSTDEF.name,
                                    CONSTDEF.value);
        };
   }

   rule Constdef {
        CONSTDEF ::= CONSTNAME = #Number compute {
             CONSTDEF.name = CONSTNAME.name;
             CONSTDEF.value = Integer.parseInt(#Number.value());
          };
   }
}
```

a) `EXPRESSION.ok = CONSTPART.ok;`
b) `PROGRAM.value =  EXPRESSION.value;`
c) `EXPRESSION.env = CONSTPART.env;`
d) `PROGRAM.ok = EXPRESSION.ok && CONSTPART.ok;`
e) We need all the semantic equations.