UNIVERSITETET I BERGEN

KANDIDAT

135

PRØVE

INF222 0 Programmeringsspråk

Emnekode	INF222
Vurderingsform	Skriftlig eksamen
Starttid	23.09.2025 09:00
Sluttid	23.09.2025 12:00
Sensurfrist	--
PDF opprettet	10.10.2025 10:53

Exam Information

Oppgave	Tittel	Status	Poeng	Oppgavetype
ℹ	INF222 exam 23.9.2025			Informasjon eller ressurser

Type systems

Oppgave	Tittel	Status	Poeng	Oppgavetype
1.1	Typing disciplines	Riktig	7/7	Paring
1.2	Subtyping relations (Java / Kotlin)	Delvis riktig	4/5	Sammensatt
1.3	Wildcards (Java)	Delvis riktig	2/3	Sammensatt
1.4	Type erasure (Java)	Delvis riktig	4/4.5	Sammensatt
1.5	Variance (Kotlin)	Riktig	10.5/10.5	Sammensatt
1.6	Formal semantics: Java interfaces and classes	Ubesvart	4/4	Muntlig

Lifetimes, ownership, borrowing, scoping, passing modes

Oppgave	Tittel	Status	Poeng	Oppgavetype
2.1	Parameter passing modes	Delvis riktig	4/7	Sammensatt
2.2	Ownership and borrowing (Rust)	Delvis riktig	8/10	Sammensatt
2.3	Pointers (Pascal)	Riktig	4.5/4.5	Sammensatt

Aspect-Oriented Programming

Oppgave	Tittel	Status	Poeng	Oppgavetype
3.1	AspectJ	Riktig	5/5	Sammensatt

Language design

Oppgave	Tittel	Status	Poeng	Oppgavetype
4.1	Language design	Besvart	5.5/8	Programmering
4.2	Concrete syntax (grammar)	Delvis riktig	10.5/12.5	Sammensatt

Parsers and Interpreters

Oppgave	Tittel	Status	Poeng	Oppgavetype
5.1	Interpreters (Haskell)	Riktig	12/12	Sammensatt
5.2	Formal grammars and calculation of set "First"	Delvis riktig	6/7	Sammensatt

1.1 Typing disciplines

Match typing disciplines (given in the columns) and their descriptions (given in the rows).

	dynamically typed	nominally typed	dependently typed	statically typed	strongly typed	(some other typing discipline)
types are known at compile time	○	○	○	◉ ✓	○	○
types like "string of length 100" or "integer whose value is less than 500" are possible	○	○	◉ ✓	○	○	○
type checking occurs during program execution	◉ ✓	○	○	○	○	○
all operations are permitted on values of all types	○	○	○	○	○	◉ ✓
rules are enforced to ensure that operations are performed only on compatible types	○	○	○	○	◉ ✓	○
type compatibility is based on explicit type declarations and/or names	○	◉ ✓	○	○	○	○
variables are implicitly coerced to	○	○	○	○	○	○

	dynamically typed	nominally typed	dependently typed	statically typed	strongly typed	(some other typing discipline)
different types						

Maks poeng: 7

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

9 3 5 2 6 3 8

1.2 Subtyping relations (Java / Kotlin)

Assume that `Coin` is a subtype of `Money`.
For each of the type combinations in the table below, determine the relationship between them.
Pay attention to the language mentioned in the leftmost column.

Java:	`Collection<W>`	is-SUPERtype-of ✓ (is-SUBtype-of, is-SUPERtype-of, is-not-related-to, is-the-same-as)	`List<W>`
Java:	`List<? super Coin>`	is-SUPERtype-of ✗ (is-SUBtype-of, is-SUPERtype-of, is-not-related-to, is-the-same-as)	`List<?>`
Java:	`List<Money>`	is-not-related-to ✓ (is-SUBtype-of, is-SUPERtype-of, is-not-related-to, is-the-same-as)	`List<Coin>`
Java:	`Set<Money>`	is-SUBtype-of ✓ (is-SUBtype-of, is-SUPERtype-of, is-not-related-to, is-the-same-as)	`Collection<Money>`
Kotlin:	`List<Coin>`	is-SUBtype-of ✓ (is-not-related-to, is-SUBtype-of, is-SUPERtype-of, is-the-same-as)	`List<Money>`

Maks poeng: 5

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

1 4 6 6 5 4 1

1.3 Wildcards (Java)

Consider the Java code below. In this code, there are three lines that attempt to perform operations on `list`. For each of those lines, determine whether the operation is valid or not (i.e., *will the code with that line successfully compile, or will the compiler report an error?*)

Hint: Recall the Get-and-Put principle and how it works for the `**extends**` and `**super**` wildcards.

```
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
public class Main {
    public static void main(String[] args) {

        List<? extends Number> list = new ArrayList<>(Arrays.asList(1, 2, 3));

        // reverse the collection


        Collections.reverse(list);  // this is allowed -- this line will be successfully compiled
(this is allowed -- this line will be successfully compiled, this is NOT allowed -- the compiler will report an error about this line)

        // then remove the 0-th element


        list.remove(0); // this is NOT allowed -- the compiler will report an error about this line
 (this is allowed -- this line will be successfully compiled, this is NOT allowed -- the compiler will report an error about this line)

        // and then add `null` to the list


        list.add(null); // this is allowed -- this line will be successfully compiled (this is allowed -- this line will be successfully compiled, this is NOT allowed -- the compiler will report an error about this line)

    }
}
```

Maks poeng: 3

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

7 9 1 9 8 7 1

1.4 Type erasure (Java)

Fill in the table below:

type that uses a generic parameter	is erased to what type
`T`	`Object`
`Iterable<? extends T>`	Iterable ✓
`List<ArrayList<String>>`	List ✓
`List<T>[]`	List[] ✓
`T[]`	Object[] ✓
`Map<?, T>`	Map ✓
`List<?>[]`	List[] ✗ (List<?>[])
`List<? extends Object>`	List ✓
`Comparable<? super T>`	Comparable ✓
`Object[]`	Object[] ✓

Maks poeng: 4.5

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

9 7 5 3 2 0 2

1.5 Variance (Kotlin)

Consider the Kotlin code in the attached PDF. Determine the variance of each of the generic parameters.

generic parameter **T** is ...	covariant ✓ (invariant, covariant, contravariant)
generic parameter **U** is ...	invariant ✓ (invariant, covariant, contravariant)
generic parameter **V** is ...	covariant ✓ (invariant, covariant, contravariant)

Hence, we can rewrite the interface declaration as follows:

`interface FancyInterface<` out T ✓ (in T, out T, T) `,` U

✓ (in U, out U, U) `,` out V ✓ (in V, out V, V) `>`

Assume that:

- `Kitten` is a subtype of `Cat`
- `Cat` is a subtype of `Animal`
- `Dog` is a subtype of `Animal`
- `Turtle` is a subtype of `Animal`

This means that:

`FancyInterface<Cat,Dog,Turtle>` is not related to ✓ (is equivalent to, is not related to, is a SUPER-type of, is a SUB-type of) `FancyInterface<Kitten,Dog,Animal>`.

Maks poeng: 10.5

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

4 3 8 0 3 4 6

1.6 Formal semantics: Java interfaces and classes

The attached PDF mentions three inference rules that specify some of the semantics of Java classes and interfaces and the relationships between them.

The first rule is of a technical nature and is only given here for the context. It introduces the notion of an interface being implemented by a class (this is denoted by "$c \sqsubset_{implements} i$" in the conclusion of the rule).

Your task is to consider inference rule 2 and inference rule 3, and fill in the missing parts of these rules.

- First, you need to be able to explain (for yourself) what each rule is trying to specify intuitively. You DO NOT need to write this explanation in the answer.
- Then, you need to fill in what is missing in the rules, using a notation that is compatible with the rest of the rules. Please avoid "inventing" your own notation here.

The exam organizers should have given you a special piece of paper for writing the answer for this question. If you don't have this piece of paper, please raise your hand and ask one of the exam organizers to give you this paper. Don't forget to fill in all the needed details (e.g., course code INF222, date of the exam 23.9.2025, candidate ID, unique question code, etc.).

In your answer for this task, you should only write down the inference rules 2 and 3, with the missing parts filled in by you. This means that on the piece of paper, there should be written two complete inference rules (i.e., inference rule 2 and inference rule 3).

For you to get the full score for this task, the rules have to be absolutely correct.

Maks poeng: 4

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

4043051

Håndtegning 1 av 1

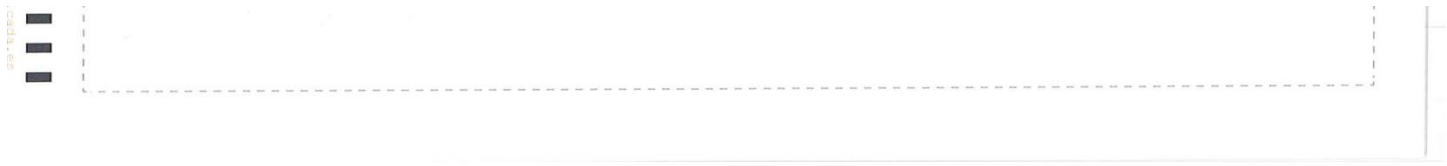
Fyll inn oppgavekode og emneinformasjon på alle skissearkene Fill out question code and test information on every sheet

Oppgavekode Question code	Dato Date	Emnekode Subject code	Kandidatnummer Candidate number	Oppgavenummer Question number	Sidetall Page number
404051	9/23/2025	INF222	135	1.6	1/1 av/of

Tegneområde Drawing area

Rule 2) $\dfrac{\Gamma \vdash C \text{ implements } i_1 \quad \Gamma \vdash i_1 \text{ extends } i_2}{\Gamma \vdash C \text{ implements } i_2}$

Rule 3) $\dfrac{\Gamma \vdash c_1 \text{ class } c_2 \quad \Gamma \vdash c_2 \text{ implements } i}{\Gamma \vdash c_1 \text{ implements } i}$



2.1 Parameter passing modes

Consider the code snippet in the attached PDF. This code snippet is written in some imaginary programming language that supports specifying parameter passing modes. Select a passing mode for each of the parameters of the methods.

Here is the same code as in the attached PDF, where you can select passing modes for each parameter.

interface PriorityQueue {

method insert(obs ✓ (obs, upd, out)Element e, obs ✓

(obs, upd, out)Priority p, upd ✓ (obs, upd, out)PriorityQueue q);

method findMin(upd ✗ (obs, upd, out)Element e, obs ✓
(obs, upd, out)PriorityQueue q);

method extractMin(upd ✗ (obs, upd, out)Element e, out ✗
(obs, upd, out)PriorityQueue q);

method size(upd ✗ (obs, upd, out)int n, obs ✓ (obs, upd,
out)PriorityQueue q);

method buildFrom(upd ✗ (obs, upd, out)PriorityQueue q, obs
✓ (obs, upd, out)Pair<Element,Priority>[] items);

method remove(obs ✓ (obs, upd, out)Element e, upd ✗

(obs, upd, out)boolean b, upd (obs, upd, out)PriorityQueue q);

}

Maks poeng: 7

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

7 2 8 6 8 6 8

2.2 Ownership and borrowing (Rust)

Fill in the blanks in the Rust code below.

```
fn main() {
    let mut x = 10;

    let y = [ &mut x ] ✓ (mut x, &mut x, x, &x);

    println!("y = {}", [ y ] ✓ (&mut y, y, mut y));
    y.add_assign(1);

    println!("y = {}", [ y ] ✓ (y, &mut y, mut y));
    let a = String::from("hello");

    let b = [ &a ] ✓ (a, mut a, &mut a, &a);
    println!("b = {}", b);
    println!("a = {}", a);

    let c = [ &mut a ] ✗ (a, &mut a, mut a);
    println!("c = {}", c);
}
```

Maks poeng: 10

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

5 0 6 5 2 4 6

2.3 Pointers (Pascal)

Consider the Pascal code in the attached PDF.
What will this program print?

line ...	... will print
`writeln(x);`	20 ✓
`writeln(p^);`	30 ✓
`writeln(q^);`	20 ✓

Maks poeng: 4.5

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

9 8 2 2 8 8 6

3 AspectJ

Fill in the blanks in the following AspectJ code.
The join point is execution of methods, and the pointcut is all public methods of class `Person` (i.e., any public method of class `Person`, and the name of the method can be anything, its return type can be anything, and it can have any amount of parameters).

```
public aspect PerformanceAspect {

    pointcut publicOperation() : [execution] (args, handler, set, execution, this, get)
(public [* Person.*(..)] );

    Object [around] (before, around, after)() : publicOperation() {
        long start = System.nanoTime();

        Object ret = [proceed()] (after(), around(), null, this, proceed(), before());
        long end = System.nanoTime();
        System.out.println(
            "TIME: method " +
            thisJoinPointStaticPart.getSignature(). getName() +
            " took " + (end-start) + " nanoseconds");
        return ret;
    }
}
```

Maks poeng: 5

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

0 4 5 9 1 1 7

4.1 Language design

Consider the following sample code written in a programming language. The language is tailored for describing printing commands for a sticker printer:

```
sticker CocaCola {
    print "text1" in bold
    print "text2" in bold + italic
    print "text3" in bold + green
    print "text4" in bold + large
}
```

The code above has four printing statements. Each statement specifies a string to be printed and also a style of the font.

One problem with this language is that it has so-called **"repetition viscosity"**. (*The term "viscosity" comes from mechanics; informally you can think of it via this analogy: water has low viscosity -- when you pour it, it flows "fast", but honey has high viscosity -- when you pour it, it flows very "slowly", because it's "thick").*

In the example above, the four printing statements specify different font styles, but there is a commonality between these four styles, namely, they are all using bold font. For the user of the language, this manifests in the need of having repetitive specifications. Imagine now that the user decides to change `bold` to `underline` (while keeping the rest of the styles intact) -- this would require erasing the word "bold" and replacing it with word "underline" four times.

Suggest another syntax for this language, where modifying some part of the font style would only require doing such a modification in a single place.

In your answer, write your version of the code above. You don't need to define a grammar, etc. -- ***you only need to rewrite the example above using another syntax which would be free of***

repetition viscosity.
Please also write a very short explanation on how you came up with the design you came up with.

Write your answer in the box below. Changes are saved automatically.

```
type format {
    bold,
    italic,
    green,
    large,
    underline
}

//part of standard lib that has one void, printAll. whitch takes a list texts
    whitch is a list of groups. a group has a name and a list of texts. It also
    takes a global format to apply to all, and group specific formats

printer {
    printAll (texts: [groupIdent:[text]] globalFormat: [format] formats: [
        (groupIdent:[format])]){
        for (groupIdent, text in texts){
            formats = [globalFormat]
            formats.addIfExists(formats[groupIdent])
            print(text, formats)
        }
    }
}

sticker CocaCola extends abstractStickerPrinter {
    texts: = [a:["text1"], b:["text2"], c:["text3"], d:["text4"]
    formats = [b:[italic],c:[green],d:[large]

    printer.printAll([texts, [bold] formats])
}

So the idea here is readablility, orthogonality where every format in the type
    format will work as expected.You have a function from printer called printAll
    (stdLib) which takes a group of text, a global format and a group of formats.
    then every text in every group will have the global format but only the extra
    formats from their matched group. For me this works much better where it is
    easy to change things without having to do a bunch of modifications. Dont know
    if it was nessasary to write the printer class, but that is how i would expect
    the flow to be.


```

Maks poeng: 8

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

6 1 5 8 3 5 2

4.2 Concrete syntax (grammar)

Consider a code sample in the attached PDF. Below is a grammar that defines the concrete syntax of this language. Fill in the blanks in the grammar specification.

Cheat sheet:

- `{something}*` denotes repetition (any number of times)
- `{something}+` denotes repetition (1 or more times)
- `{something}?` denotes optionality
- `something1 | something2` denotes alternatives

VariableDeclaration :
"***variable***" ident {, ident}* ({"," ident}*, {',' ident}*, {`,` ident}*) //
allow any amount of identifiers
;

ClassDeclaration :
"***class***" ident
{ MemberDeclaration **}***
"***end***" "***class***"
;

MemberDeclaration
: FieldDeclaration
| ConstructorDeclaration
| MethodDeclaration
;

FieldDeclaration :

"field" ident "***[***" VisibilityModifier ({VisibilityModifier}+, VisibilityModifier) "***]***"
;

VisibilityModifier:
"***public***" | "***private***"
;

ParametersDeclaration

: "***takes***" ident (ident {"," ident}*, ident {',' ident}*, ident {`,` ident} *)
| "***nothing***"
;

MethodDeclaration :
"**method**" ident "***[***" VisibilityModifier "***]***"
ParametersDeclaration
Statement
"**end**" "**method**"
;

ConstructorDeclaration :
"***constructor***"
ParametersDeclarations
Statement
"***end***" "***constructor***"
;

Statement
: AssignmentStatement
| IfStatement

| BlockStatement (MethodDeclaration, ParametersDeclaration, BlockStatement, ThisSpecifier, Statement)
;

BlockStatement :

{Statement}* ({Statement}*, Statement, {Statement}?)
;

AssignmentStatement :

{ ThisSpecifier **}** ? **(?, +, *)** ident "="

Expr (Expr, Statement)
;

ThisSpecifier : "***my***" ;

IfStatement :

"***if***" Expr (Statement, Expr, {Expr}?, {Statement}?)
"***then***"
Statement
"***end***" "***then***"
"***else***"
Statement
"***end***" "***else***"
"***end***" "***if***"
;

Program :

{ VariableDeclaration **|** ClassDeclaration **}+**
;

Maks poeng: 12.5

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

5 9 8 0 3 7 8

5.1 Interpreters (Haskell)

Consider the following Haskell code that implements an interpreter for a simple language.
The function `*execute*` takes a statement and an environment, and returns a new environment after evaluating the statement according to its semantics.
Fill in the blanks in the code.

```
module Main where

data Stmt
  = Skip
  | Assignment String Expr
  | Seq Stmt Stmt
  | If Expr [Stmt Stmt]
  | While [Expr Stmt]
  deriving (Show)

data Expr
  = IntConst Int
  | BoolConst Bool
  | VarUse String
  | Binary BOp [Expr Expr]
  deriving (Show)

data BOp
  = ADD
  | SUB
  deriving (Show)

data Val
  = VB Bool
  | VI Int
  | VU -- undefined
  deriving (Show, Eq)

type Environment = [ (String, Val) ]

execute :: Stmt -> [Environment] -> Environment

execute (Skip) env = [env]
```

execute (Assignment name expr) env = (name , evaluate expr env) : env

execute (Seq s1 s2) env = (execute s2 . execute s1) env

execute (If cond trueBranch falseBranch) env = **case** evaluate cond env **of**
VB True -> execute trueBranch env
VB False -> execute falseBranch env

execute (While cond loopBody) env =
execute
(
If
(cond)
(Seq
(loopBody)
(While cond loopBody)
)
(Skip)
)
env

Maks poeng: 12

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

9 0 7 8 4 1 1

5.2 Formal grammars and calculation of set "First"

Consider the following grammar:

$S \to aSb \mid CB$
$B \to bB \mid \varepsilon$
$C \to b \mid \varepsilon$
$Q \to a \mid b \mid \varepsilon$

Recall that ε means an empty string.

Based on this grammar, do the following 3 subtasks.

Subtask 1: Determine the values of the set $\mathbf{First}$ for the following:

Notes:

- *If you need to type ε in one of your answers, then please type* **epsilon** *(see example below in the table).*
- *If you need to type more than one symbol into a slot, then separate them with a comma*
  - *For example:* ***a,b***

$\mathbf{First}(S \to CB)$	b, epsilon ✓
$\mathbf{First}(B \to \varepsilon)$	**epsilon**
$\mathbf{First}(Q)$	a,b,epsilon ✓
$\mathbf{First}(C)$	b, epsilon ✓
$\mathbf{First}(B)$	b, epsilon ✓
$\mathbf{First}(S)$	s, b, epsilon ✗ (a,b,epsilon, a,epsilon,b, b,epsilon,a, b,a,epsilon, epsilon,a,b, epsilon,b,a)

Subtask 2: Based on your calculation of the sets $\mathbf{First}$ above: is this grammar LL(1) or not?, i.e., *is it possible to define a recursive descent parser for it?*

Select the correct answer:

- (●) Yes, this grammar is an LL(1)-grammar ✓
- ( ) No, this grammar is NOT an LL(1)-grammar

Subtask 3: Which of the nonterminal symbols can be safely removed from the grammar?
(*safe removal means that the language generated by the grammar will <u>not</u> change as a result of such removal*)

Select the correct answer:

- ( ) S
- ( ) B
- ( ) C
- (●) Q ✓
- ( ) None of the nonterminal symbols can be safely removed from the grammar

Maks poeng: 7

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

0 5 7 9 3 2 5