UNIVERSITETET I BERGEN

KANDIDAT

139

PRØVE

INF222 0 Programmeringsspråk

Emnekode	INF222
Vurderingsform	Skriftlig eksamen
Starttid	22.05.2025 09:00
Sluttid	22.05.2025 12:00
Sensurfrist	--
PDF opprettet	27.05.2025 15:33

Exam Information

Oppgave	Tittel	Status	Poeng	Oppgavetype
i	INF222 exam 22.5.2025			Informasjon eller ressurser

Type systems

Oppgave	Tittel	Status	Poeng	Oppgavetype
1.1	Typing disciplines	Delvis riktig	4/7	Paring
1.2	Subtyping relations (Java / Kotlin)	Riktig	5/5	Sammensatt
1.3	Wildcards (Java)	Delvis riktig	3/5	Sammensatt
1.4	Type erasure (Java)	Feil	0/4	Sammensatt
1.5	Variance (Kotlin)	Riktig	9/9	Sammensatt

Lifetimes, ownership, borrowing, scoping, passing modes

Oppgave	Tittel	Status	Poeng	Oppgavetype
2.1	Parameter passing modes	Delvis riktig	8/9.5	Sammensatt
2.2	Lexical vs. dynamic scoping	Riktig	4/4	Sammensatt
2.3	Ownership and borrowing (Rust)	Riktig	10/10	Sammensatt
2.4	Variable lifetimes and aliasing (Pascal)	Riktig	4/4	Sammensatt

Aspect-Oriented Programming

Oppgave	Tittel	Status	Poeng	Oppgavetype

3.1	AspectJ	Delvis riktig	1/5	Sammensatt

Language design

Oppgave	Tittel	Status	Poeng	Oppgavetype
4.1	Language design criteria	Delvis riktig	2/7	Sammensatt
4.2	Concrete syntax (grammar)	Riktig	11.5/11.5	Sammensatt

Parsers and Interpreters

Oppgave	Tittel	Status	Poeng	Oppgavetype
5.1	Interpreters (Haskell)	Delvis riktig	4/12	Sammensatt
5.2	Formal grammars and calculation of set "First"	Delvis riktig	4/7	Sammensatt

1.1 Typing disciplines

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

	(some other typing discipline)	dependently typed	statically typed	dynamically typed	nominally typed	strongly typed
variables are implicitly coerced to different 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	○	○	○	○	● ✓	○
types are known at compile time	○	○	● ✓	○	○	○
types like "array with 42 elements" are possible	● ✗	○ ✓	○	○	○	○
type checking occurs during program execution	○	○	○	● ✓	○	○
all operations are permitted on values of all types	○ ✓	○	○	○	○	○

Maks poeng: 7

1.2 Subtyping relations (Java / Kotlin)

Assume that `Cat` is a subtype of `Animal`.

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:	`List<E>`	is-SUBtype-of ✓ (is-SUBtype-of, is-SUPERtype-of, is-not-related-to, is-the-same-as)	`Collection<E>`
Java:	`List<? extends Cat>`	is-SUBtype-of ✓ (is-SUBtype-of, is-SUPERtype-of, is-not-related-to, is-the-same-as)	`List<?>`
Kotlin:	`List<Cat>`	is-SUBtype-of ✓ (is-SUBtype-of, is-SUPERtype-of, is-not-related-to, is-the-same-as)	`List<Animal>`
Java:	`List<Animal>`	is-not-related-to ✓ (is-SUBtype-of, is-SUPERtype-of, is-not-related-to, is-the-same-as)	`List<Cat>`
Java:	`Collection<Animal>`	is-SUPERtype-of ✓ (is-SUBtype-of, is-SUPERtype-of, is-not-related-to, is-	`Set<Animal>`

		the-same-as)		

Maks poeng: 5

1.3 Wildcards (Java)

Consider the Java code below. In this code, there are five 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?*)

```
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));
```

list.get(0); // 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)

list.remove(0); // 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)

list.add(10); // 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)

list.add(null); // 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)

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)

```
   }
}
```

Maks poeng: 5

1.4 Type erasure (Java)

Consider the following Java code:

```
public static <T extends Object & Comparable<? super T>> T max(Collection
    // business logic here
}
```

What will be the signature of this method after the type erasure?

public static **Object max(Collection<Object> coll)** ✗

(Object max(Collection coll)) **{**

// business logic here

}

Maks poeng: 4

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

Hence, we can rewrite the interface declaration as follows:

`interface StrangeThing<` out T ✓ (in T, out T, T) `,` in U ✓ (in U, out U, U) `,` out V ✓ (in V, out V, V) >

Maks poeng: 9

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 BinarySearchTree {

method insert(obs ✓ (obs, upd, out)Element e, upd ✓ (obs, upd, out)Tree t);

method search(out ✓ (obs, upd, out)boolean b, obs ✓ (obs, upd, out)Element e, obs ✓ (obs, upd, out)Tree t);

method findMin(upd ✗ (obs, upd, out)Element e, obs ✓ (obs, upd, out)Tree t);

method height(out ✓ (obs, upd, out)int h, obs ✓ (obs, upd, out)Tree t);

method isBalanced(out ✓ (obs, upd, out)boolean b, obs ✓ (obs, upd, out)Tree t);

method copy(upd ✗ (obs, upd, out)Tree copyT, obs ✓ (obs, upd, out)Tree t);

method merge(upd ✓ (obs, upd, out)Tree t1, obs ✓ (obs, upd, out)Tree t2);
}

Maks poeng: 9.5

2.2 Lexical vs. dynamic scoping

Consider the code in the attached PDF.
This code is written in some imaginary programming language called *WeirdLanguage*.

The value printed by the program in the attached PDF will depend on whether *WeirdLanguage* uses lexical or dynamic scoping. For the values given below, determine which scoping the language uses.

assuming that ...	the value printed by the program will be ...
WeirdLanguage uses dynamic ✓ **(dynamic, both lexical and dynamic, lexical)** scoping	8
WeirdLanguage uses lexical ✓ **(dynamic, both lexical and dynamic, lexical)** scoping	14

Maks poeng: 4

2.3 Ownership and borrowing (Rust)

Fill in the blanks in the Rust code below.

fn main() {

let /*nothing is needed here*/ ✓ (/*nothing is needed here*/, &, &mut)x = 10;

let y = & ✓ (both & and &mut are OK, &, &mut)x;
println!("y = {}", y);
{

let z = & ✓ (&, &mut, both & and &mut are OK)x;
println!("z = {}", z);
}
let mut a = 5;
{

let b = &mut ✓ (/*nothing is needed here*/, &mut, &, mut) a;
b.add_assign(1); *// increments the value of `b`*
}
println!("a = {}", a);

let c = both & and &mut are OK ✓ (&mut, both & and &mut are OK, &)a;
println!("c = {}", c);
}

Maks poeng: 10

2.4 Variable lifetimes and aliasing (Pascal)

Consider the code in the attached PDF.

A developer has run this Pascal program and did not get the value 42 printed.
Why did this happen?

Select the correct reason / reasons:

- ☐ The compiler detects that `x` has a short lifetime and replaces its value with random bytes for safety reasons.
- ☐ Pascal pointers *require* explicitly extended lifetimes via `new`, and since `ptr` was not allocated this way, dereferencing it causes undefined results.
- ☐ The lifetime of `ptr` is tied to that of `x`, and since `ptr` is copied out of scope, it becomes a `null` reference when `x` is destroyed.
- [✓] The function `CreateAlias` returns a pointer to a local variable whose lifetime ends when the function exits, so `alias^` refers to invalid memory and the result is undefined. ✓

!!! IMPORTANT !!!

How are points calculated for this Task 2.4? (The explanation below is only applicable for this Task 2.4.)

- If you answer this task absolutely correctly, you will get 4 points for this task.
- **<u>If you answer this task incorrectly, you get will -2 (minus two) points, and those negative points will affect the total amount of points that you get for the entire exam.</u>**
- **If you skip this task and do not answer anything, then you will get 0 points for this task.**

Maks poeng: 4

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 (i.e., any public method of any class, 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 MeasureTimeAspect {

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

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

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

Maks poeng: 5

4.1 Language design criteria

Consider the 4 code samples written in some imaginary language in the attached PDF.

Each of the samples has a design flaw that violates orthogonality of the language design. Those flaws are explained in the comments starting with '*// Error*'.

Determine the type of orthogonality violation for each of the samples.

Sample 1 violates	combination orthogonality ✗ (sort orthogonality, number orthogonality, combination orthogonality)
Sample 2 violates	combination orthogonality ✓ (number orthogonality, sort orthogonality, combination orthogonality)
Sample 3 violates	sort orthogonality ✗ (number orthogonality, combination orthogonality, sort orthogonality)
Sample 4 violates	number orthogonality ✗ (sort orthogonality, combination orthogonality, number orthogonality)

Maks poeng: 7

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}* ✓
;

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

MemberDeclaration
: FieldDeclaration
| ConstructorDeclaration
| MethodDeclaration ✓
;

FieldDeclaration :

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

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

ConstructorDeclaration :
"***constructor***"
ParametersDeclaration
Statement
"***end***" "***constructor***"

;

ParametersDeclaration

: "***takes***" ident {"," ident}* ✓

| "***nothing***"

;

MethodDeclaration :

"**method**" ident "***[***" VisibilityModifier ✓ "***]***"

ParametersDeclaration

Statement

"**end**" "**method**"

Statement

: AssignmentStatement

| IfStatement

| BlockStatement ✓ (MethodDeclaration, SelfSpecifier, ParametersDeclaration, Statement, BlockStatement)

;

BlockStatement :

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

;

AssignmentStatement :

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

Expr ✓ (Expr, Statement)

;

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

IfStatement :

"***if***" Expr ✓ ({Statement}?, Statement, Expr, {Expr}?)

"***then***"

```
        Statement
     "end" "then"
     "else"
        Statement
     "end" "else"
  "end" "if"
  ;

Program :

  { VariableDeclaration | ClassDeclaration ✓ }+

  ;
```

Maks poeng: 11.5

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 Stmt, Statement Statement)
  | 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 s2 .

execute s1, flip (.) (execute s2) (execute s1), (.) (execute s2) (execute s1), (.) <$> execute s2 <*> execute s1, liftA2 (.) (execute s2) (execute s1), (liftA2 (.)) (execute s2) (execute s1), (. execute s1) (execute s2))) 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
        ( evaluate cond == VB True ✗ (cond))
        (Seq
            (loopBody)
            ( execute (While cond loopbody) env ✗ (While cond loopBody))
        )
        (Skip)
    )
    env
```

Maks poeng: 12

5.2 Formal grammars and calculation of set "First"

Consider the following grammar:

$S \to aSb \mid BC$
$B \to bB \mid \varepsilon$
$C \to d \mid a$
$Z \to a \mid b$

Recall that ε means an empty string.

Based on this grammar, do the following 4 subtasks.

Subtask 1: What does the nonterminal B define?

Select the correct answer:

- ○ terminal symbol `b` repeated any number of times, but at least one time
- ○ terminal symbol `b` repeated an even number of times
- ○ terminal symbol `b` repeated an odd number of times
- ○ just a single terminal symbol `b` or and empty string
- [✓ ● terminal symbol `b` repeated any number of times] ✓

Subtask 2: Determine the values of the set $\textbf{First}$ 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 (see example below in the table).*

$\textbf{First}(S \to BC)$	b,d,a ✓
$\textbf{First}(B \to \varepsilon)$	**epsilon**
$\textbf{First}(Z)$	**a,b**
$\textbf{First}(C)$	d,a ✓

$\mathbf{First}(B)$	b ✗ (b,epsilon, epsilon,b, b,e, e,b)
$\mathbf{First}(S)$	a,b,d,a ✗ (a,b,d, a,d,b, b,d,a, b,a,d, d,a,b, d,b,a)

Subtask 3: 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 4: 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
- [✓ Z] ✓
- None of the nonterminal symbols can be safely removed from the grammar

Maks poeng: 7