The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, AUTUMN SEMESTER 2023-2024

Introduction to Formal Reasoning

Time allowed 2 hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced

Answer ALL FOUR questions

No calculators are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a dictionary to translate between that language and English provided that neither language is the subject of this examination.

No electronic devices capable of storing and retrieving text may be used.

DO NOT turn examination paper over until instructed to do so

INFORMATION FOR INVIGILATORS: None

Question 1: Propositional logic

a. What are the propositional constants and connectives defined in Lean? Copy the table below and complete the missing entries.

Say either primitive if a connective is primitive, or give the definition.

connective	name	primitive or definition
`P ∧ Q`	conjunction, and	primitive
	disjunction, or	
`false`	falsum, false	
`¬ P`		
		`(P → Q) ∧ (Q → P)`

[10 marks]

b. Copy the following table into your answer book and complete it, indicating whether the left-to-right (→) and the right-to-left (←) implications are

I provable intuitionistically

C provable classically

X unprovable classically

Use the letters I,C,X to indicate your choice or leave the square empty, if you don't know.

Hint: Think of a concrete example, e.g. P = we go to the zoo, Q = it rains.

	→	←
`¬ (P ∧ Q) ↔ ¬ P ∧ ¬ Q`	X	I
`¬ (P ∧ Q) ↔ ¬ P ∨ ¬ Q`		
`¬ (P ∨ Q) ↔ ¬ P ∨ ¬ Q`		
`¬ (P ∨ Q) ↔ ¬ P ∧ ¬ Q`		

[10 marks]

c. How do we read `P → Q → R`?

(i) Is it `(P → Q) → R` or `P → (Q → R)`?

(ii) Are the two readings equivalent? That is can we prove

```
(P → Q) → R ↔ P → (Q → R)
```

or alternatively can we prove

```
¬ ((P → Q) → R ↔ P → (Q → R))
```

or neither?

(iii) Explain your answer (you may use Lean on paper to do that).

[5 marks]

End of Question 1: Total 25 marks

Question 2: Predicate logic

a. In Lean what are the tactics we use in predicate logic? Copy the following table into your answer book and complete it. Answer by providing tactics with their arguments.

Operator	How to prove?	How to use?
∀	`assume x`	
∃		
=		

[10 marks]

b. Given

```
constant People : Type
constants Male Female : People → Prop
constant Parent : People → People → Prop
```

where `People` is the type of people, `Male x` means `x` is male, `Female x` means `x` is female, and `Parent x y` means `x` is the parent of `y`.

(i) Define the relation `x` is the mother of `y` by replacing `sorry` in the following line:

```
def Mother(x y : People) : Prop := sorry
```

(ii) Define the relation `x` is the grandmother of `y` by replacing `sorry` in the following line:

```
def Grandmother(x y : People) : Prop := sorry
```

(iii) Translate Everybody has a mother into predicate logic by replacing `sorry` in the following line:

```
def every_mother : Prop := sorry
```

(iv) Translate Everybody has at most one mother into predicate logic by replacing `sorry` in the following line:

```
def only_one_mother : Prop := sorry
```

You can use previously defined relations in the subsequent questions.

[10 marks]

c. How do we read `∀ x : A,PP x ∧ Q`?

(i) Is it `(∀ x : A,PP x) ∧ Q` or `∀ x : A,(PP x ∧ Q)`?

(ii) Are the two readings equivalent?

(iii) Justify your answer (you may use Lean on paper to do that).

[5 marks]

End of Question 2: Total 25 marks

Question 3: Bool and natural numbers

a. Consider the following propositions:

```
P₁ = ∀ x:bool,∃ y:bool, x ≠ y
P₂ = ∃ x:bool,∀ y:bool, x ≠ y
P₃ = ∀ x y : bool, x ≠ y → ! x ≠ ! y
P₄ = ∀ x : bool, x = ! x → x = tt
P₅ = ∀ x y z : bool, x=y ∨ x=z
```

Determine whether they are provable (P) or their negation is provable (N). Copy the following table into your answer book and complete it using (N) or (P). Leave the field blank, if you don't know.
Could it happen that neither the proposition nor its negation is provable in Lean?

Proposition	(P) or (N) ?
P_1	
P_2	
P_3	
P_4	
P_5	

[10 marks]

b. We are in the following Lean proof state:

```
n: ℕ
⊢ PP n
```

What is the state after we do `induction n with n' ih,?` You may omit irrelevant details.
And what is the state after `cases n with n',?`
What is the difference?

[10 marks]

c. How do we prove that `succ` is injective,

```
∀ m n : ℕ, succ m = succ n → m = n
```

without using the `injection`-tactic?
It is enough to sketch the proof.

[5 marks]

End of Question 3: Total 25 marks

Question 4: Lists and trees

a. We defined the relation `Perm : list A → list A → Prop` (permutation) inductively as shown below (using auxilliary `Insert : A → list A → list A → Prop`).

```
inductive Insert : A → list A → list A → Prop
| ins_hd : ∀ a:A, ∀ as : list A,Insert a as (a :: as)
| ins_tl : ∀ a b:A, ∀ as as′: list A,
                    Insert a as as′ → Insert a (b :: as) (b :: as′)

inductive Perm : list A → list A → Prop
| perm_nil : Perm [] []
| perm_cons : ∀ a : A, ∀ as bs bs′ : list A,
                  Perm as bs → Insert a bs bs′ → Perm (a :: as) bs′
```

Consider the following propositions:

```
def Q₁ := Perm [1,2,3] [3,2,1]
def Q₂ := Perm [1,2] [2,2,1]
def Q₃ := ∀ xs ys : list A, ¬ (Perm xs ys) → xs ≠ ys
def Q₄ := ∀ xs ys xs′ ys′ : list A,
    Perm (xs ++ xs′) (ys ++ ys′)
    → Perm xs ys
def Q₅ := ∀ xs xs′ ys : list A,
  Perm (xs ++ xs′) (ys ++ xs′)
  → Perm xs ys
```

Determine whether they are provable (P) or their negation is provable (N). Copy the following table into your answer book and complete it using (N) or (P). Leave the field blank, if you don't know.

Could it happen that neither the proposition nor its negation is provable in Lean?

Proposition	(P) or (N) ?
Q_1	
Q_2	
Q_3	
Q_4	
Q_5	

[10 marks]

b. We define a type of boolean expressions where only variables can be negated:

```
inductive Expr : Type
| var : bool → string → Expr
| and : Expr → Expr → Expr
| or : Expr → Expr → Expr
```

The boolean parameter to `var` indicates whether the variable is negated. E.g. `x` is written as `var tt x`, while `! x` is written as `var ff x`. So for example the expression `(x && !y) || x` would be written as

```
or (and (var tt "x") (var ff "y")) (var tt "x")
```

We define a type of environments `Env` assigning booleans to variables, and an evaluator for `Expr`:

```
def Env : Type := string → bool

def eval : Expr → Env → bool
| (var b x) env := if b then env x else ! (env x)
| (and e₁ e₂) env := eval e₁ env && eval e₂ env
| (or e₁ e₂) env := eval e₁ env || eval e₂ env
```

(i) There is no constructor for negation - your task is to define a function that negates an expression:

```
def neg : Expr → Expr
```

such that the following theorem is provable:

```
theorem neg_ok : ∀ e : Expr, ∀ env : Env ,
    eval (neg e) env = ! (eval e env)
```

You are not allowed to modify `Expr` or `eval`.

(ii) How would you prove `neg_ok` in Lean? You don't need to give all the details but at least answer the following questions:

1. What properties of operations on `bool` do you rely on? Can they be proven?
2. What is the structure of the proof? What tactic you use to prove properties of `Expr` ?

[15 marks]

End of Question 4: Total 25 marks