Chapter 1

Library syntax

`Section` *syntax*.

The set of well-formed formulae of the monomodal logic K is built from an enumerable set of propositional symbols:

`Definition` *Props* := *nat*.

and a set of constants:

`Inductive` *Const*: `Set` := *tt* | *ff*.

where *tt* represent *verum* and *ff* represents *falsum*. The set of formulae is inductively defined as follows:

```
Inductive formula : Type :=
  | PropsF : Props → formula
  | ConstF : Const → formula
  | Not : formula → formula
  | And : formula → formula → formula
  | Or : formula → formula → formula
  | Imp : formula → formula → formula
  | Box : formula → formula
  | Diamond : formula → formula.
```

The size of a formula is defined in the usual way:

```
Fixpoint size (f: formula) : nat :=
  match f with
  | PropsF p ⇒ 1
  | ConstF _ ⇒ 1
  | Not g ⇒ 1 + size g
  | And f g ⇒ 1 + (size f) + (size g)
  | Or f g ⇒ 1 + (size f) + (size g)
  | Imp f g ⇒ 1 + (size f) + (size g)
  | Box f ⇒ 1 + (size f)
```

| *Diamond* $f \Rightarrow 1 + (size\ f)$
`end`.

`End` *syntax*.

Chapter 2

Library semantics

```
Require Import ModalLogic.syntax.
Require Import Relations.Relation_Definitions.
Require Import Coq.Logic.Classical_Prop.

Section semantics.
```

The semantics of the monomodal logic K is based on Kripke Models, where we have a non-empty set of worlds:

```
Definition W := Type.
```

and a relation over the set of worlds:

```
Definition R := relation W.
```

from which a frame, there is an ordered pair of worlds and a relation, can be defined:

```
Definition Frame := (W × R) % type.
```

The evaluation function assigns a Boolean to a world and a propositional symbol:

```
Definition pi := W → Props → bool.
```

A Kripe model (or structure) is a frame equiped with a valuation function:

```
Definition Model := (Frame × pi) %type.
```

The satisfiability relation is recursively defined as follows:

```
Fixpoint sat (M:Model) (w:W) (f:formula): Prop :=
 let R := snd(fst(M)) in
 let pi:= snd M in
  match f with
  | ConstF tt ⇒ True
  | ConstF ff ⇒ False
  | PropsF p ⇒ if (pi w p) then True else False
  | Not f ⇒ ~(sat M w f)
  | And f g ⇒ (sat M w f) ∧ (sat M w g)
  | Or f g ⇒ (sat M w f) ∨ (sat M w g)
```

| *Imp f g* ⇒ ~(*sat M w f*) ∨ (*sat M w g*)
| *Box f* ⇒ ∀ *w'*:*W*, (*R w w'*) → (*sat M w' f*)
| *Diamond f* ⇒ ∃ *w'*:*W*, (*R w w'*) ∧ (*sat M w' f*)
`end`.

Note in the above definition that the metatheoretical connectives are classical.

The next definition says that a formula is locally satisfiable if there is a model and world that satisfies the formula according to the satisfaction relation just defined above.

`Definition` *local_sat* (*f*:*formula*): `Prop` := ∃ *M*:*Model*, ∃ *w*:*W*, *sat M w f*.

Two formula are semantically equivalent if the satisfaction relation coincide for every model and world:

`Definition` *sem_equiv* (*f g* : *formula*) : `Prop` :=
∀ (*M*:*Model*) (*w*:*W*), *sat M w f* = *sat M w g*.

`End` *semantics*.

Chapter 3

Library nnf

```
Require Import ModalLogic.syntax.
Require Import ModalLogic.semantics.

Require Import FunInd.
Require Import Recdef.
Require Import Psatz.

Require Import Coq.Arith.Lt.
Require Import Coq.Logic.Classical_Prop.
Require Import Coq.Logic.Classical_Pred_Type.

Section NNF.
```

The negation normal form (NNF) takes the usual form: only constants, literals (i.e. propositions or their negations), conjunctions, disjunctions and modal operators are allowed. Note that negations can only be applied to propositional symbols. The following function returns whether a formula is in NNF.

```
Fixpoint is_NNF (f:formula): Prop :=
  match f with
  | PropsF p ⇒ True
  | ConstF ff ⇒ True
  | ConstF tt ⇒ True
  | Not (PropsF p) ⇒ True
  | Not _ ⇒ False
  | And f g ⇒ (is_NNF f ) → (is_NNF g)
  | Or f g ⇒ (is_NNF f ) → (is_NNF g)
  | Imp _ _ ⇒ False
  | Box f ⇒ (is_NNF f)
  | Diamond f ⇒ (is_NNF f)
  end.
```

The next definition takes a formula and returns its NNF. This is defined as general function, because the recursion is not applied to subformulae. For instance, when taking

the transformation of an implication, one of the resulting disjuncts is the negation of its antecedent, which is not a subformula of the implication. Thus, we cannot use structural induction when writing proofs about the NNF a formula. However, the recursion (and, later, our proofs) takes smaller arguments than the formulae it is applied. We take, therefore, the size of a formula as a measure for termination.

```
Function NNF (f:formula) {measure size f}: formula :=
  match f with
  | PropsF p ⇒ PropsF p
  | ConstF ff ⇒ ConstF ff
  | ConstF tt ⇒ ConstF tt
  | Not (PropsF p)=> Not (PropsF p)
  | Not (ConstF ff) ⇒ ConstF tt
  | Not (ConstF tt) ⇒ ConstF ff
  | Not (Not f) ⇒ NNF f
  | Not (And f g) ⇒ Or (NNF (Not f)) (NNF (Not g))
  | Not (Or f g) ⇒ And (NNF (Not f)) (NNF (Not g))
  | Not (Imp f g) ⇒ And (NNF f) (NNF (Not g))
  | Not (Box f) ⇒ Diamond (NNF (Not f))
  | Not (Diamond f) ⇒ Box (NNF (Not f))
  | And f g ⇒ And (NNF f) (NNF g)
  | Or f g ⇒ Or (NNF f) (NNF g)
  | Imp f g ⇒ Or (NNF (Not f)) (NNF g)
  | Box f ⇒ Box (NNF f)
  | Diamond f ⇒ Diamond (NNF f)
  end.
```

The proof that the above function is indeed terminating basically relies on properties of inequalities and are automatically obtained via the procedures implemented in coq for linear arithmetic.

The first theorem shows that the transformationn into the normal form is correct, that is, that the function does produce a formula into the desired formula.

```
Theorem NNF_is_NNF (f:formula) :
  is_NNF (NNF f).
```

The next theorem shows that the formulae produced by the transformation function preserves meaning, that is, the resulting formula is semantically equivalent to the original one. The proof is by induction on the NNF of a formula. It is rather long, but it is simple: it basically relies on the induction hipothesis to show that every operation over the existing operators preserve satisfiability. Note the use of classical facts whenever needed, as the metatheory for this logic is classical.

```
Theorem eq_f_NNF_f (f : formula) :
  ∀ (M:Model) (w:W), sat M w f ↔ sat M w (NNF f).

End NNF.
```