```
module basic where

open import Agda.Builtin.Nat
  renaming (_<_ to _<ᵇ_)
  renaming (_==_ to _=n_)
open import Data.Nat.DivMod
open import Data.Sum.Base
  using(_⊎_ ; inj₁ ; inj₂)
open import Data.Nat.Base
  using (_<_)
  using (_>_)
  using (z≤n)
  using (s≤s)
  using (_≤_)
open import Data.Nat.Properties
  using (+-comm)
  using (<-trans)
  using (<-cmp)
  using (≤-refl)
open import Agda.Builtin.Equality
open import Data.Bool
  hiding (not)
  hiding (_≤_)
  hiding (_<_)
open import Data.Char
  renaming (_==_ to _=c_)
  renaming (_<_ to _<c_)
  renaming (show to show-char)
open import Data.String
  renaming (length to length-string)
  renaming (show to show-string)
  renaming (_<_ to _<s_)
  renaming (_==_ to _=s_)
  renaming (_++_ to _++s_)
open import Data.List
  renaming (lookup to lookup-list)
  renaming (all to all-list)
  renaming (or to disj)
  renaming (and to conj)
  renaming (concat to concat-list)
open import Relation.Nullary
open import Data.Product
  renaming (map to map2)
open import Data.Unit
  using (⊤)
```

```
  using (tt)
open import Data.Maybe
  renaming (_»=_ to _?>=_)
  renaming (map to map?)
open import Data.Nat.Show
open import Data.Empty
open import Relation.Nullary.Decidable
  using (toWitness)
open import Relation.Binary.Definitions
  using (tri<)
  using (tri≈)
  using (tri>)


- Basic Logic

postulate LEM : (A : Set) → Dec A
postulate FX : ∀ {A B : Set} (f g : A → B) (h : ∀ a → f a ≡ f a) → f ≡ g

intro-or-lft : {A B C : Set} → (A → B) → (A → B ⊎ C)
intro-or-lft h0 h1 = inj₁ (h0 h1)

intro-or-rgt : {A B C : Set} → (A → C) → (A → B ⊎ C)
intro-or-rgt h0 h1 = inj₂ (h0 h1)

_↔_ : Set → Set → Set
A ↔ B = (A → B) × (B → A)

and-symm : ∀ {A B : Set} → (A × B) → (B × A)
and-symm (h , g) = g , h

or-elim : ∀ {A B C : Set} → A ⊎ B → (A → C) → (B → C) → C
or-elim (inj₁ x) f g = f x
or-elim (inj₂ x) f g = g x

or-elim' : ∀ {A B C : Set} → (A → C) → (B → C) → (A ⊎ B) → C
or-elim' ha hb hab = or-elim hab ha hb

ex-elim : ∀ {A B : Set} {P : A → Set} → (∃ P) → (∀ (x : A) → P x → B) → B
ex-elim (a , h0) h1 = h1 a h0

ex-elim-2 : ∀ {A B C : Set} {P : A → B → Set} →
  (∃ λ a → ∃ (P a)) → (∀ (x : A) (y : B) → P x y → C) → C
ex-elim-2 (a , (b , h0)) h1 = h1 a b h0

ex-elim-3 : ∀ {A B C D : Set} {P : A → B → C → Set} →
  (∃ λ a → ∃ λ b → ∃ λ c → (P a b c)) → (∀ a b c → P a b c → D) → D
ex-elim-3 (a , (b , (c , h0))) h1 = h1 a b c h0
```

```
ex-elim' : ∀ {A B : Set} {P : A → Set} → (∀ (x : A) → P x → B) → (∃ P) → B
ex-elim' h0 (a , h1) = h0 a h1

ex-elim-3' : ∀ {A B C D : Set} {P : A → B → C → Set} →
  (∀ a b c → P a b c → D) → (∃ λ a → ∃ λ b → ∃ λ c → (P a b c)) → D
ex-elim-3' h0 (a , (b , (c , h1))) = h0 a b c h1

elim-lem : ∀ (A : Set) {B : Set} → (A → B) → ((¬ A) → B) → B
elim-lem A h0 h1 with LEM A
... | (yes h2) = h0 h2
... | (no h2) = h1 h2

Chars : Set
Chars = List Char

pred : Nat → Nat
pred 0 = 0
pred (suc k) = k

data Functor : Set where
  nf : Nat → Functor
  sf : Chars → Functor

data Termoid : Bool → Set where
  var : Nat → Termoid false
  fun : Functor → Termoid true → Termoid false
  nil : Termoid true
  cons : Termoid false → Termoid true → Termoid true

Term = Termoid false
Terms = Termoid true

data Bct : Set where
  or  : Bct
  and : Bct
  imp : Bct
  iff : Bct

data Formula : Set where
  cst : Bool → Formula
  not : Formula → Formula
  bct : Bct → Formula → Formula → Formula
  qtf : Bool → Formula → Formula
  rel : Functor → Terms → Formula

_=*_ : Term → Term → Formula
t =* s = rel (sf ('=' ∷ [])) (cons t (cons s nil))
```

```
_∨*_ : Formula → Formula → Formula
_∨*_ = bct or

_∧*_ : Formula → Formula → Formula
_∧*_ = bct and

_→*_ : Formula → Formula → Formula
f →* g = bct imp f g

_↔*_ : Formula → Formula → Formula
f ↔* g = bct iff f g

∀* = qtf false
∃* = qtf true

⊤* = cst true
⊥* = cst false

par : Nat → Term
par k = fun (nf k) nil

tri : ∀ {A : Set} → Nat → A → A → A → Nat → A
tri k a b c m with <-cmp k m
... | (tri< _ _ _) = a
... | (tri≈ _ _ _) = b
... | (tri> _ _ _) = c

tri-intro-lem : ∀ {A : Set} {a b c : A} (k m : Nat) → (P : A → Set) →
  (k < m → P a) → (k ≡ m → P b) → (k > m → P c) → P (tri k a b c m)
tri-intro-lem k m P h0 h1 h2 with (<-cmp k m)
... | (tri< hl he hg) = h0 hl
... | (tri≈ hl he hg) = h1 he
... | (tri> hl he hg) = h2 hg

tri-eq-lt : ∀ {A : Set} {a b c : A} (k m : Nat) → (k < m) → (tri k a b c m) ≡ a
tri-eq-lt k m h with (<-cmp k m)
... | (tri< hl he hg) = refl
... | (tri≈ hl he hg) = ⊥-elim (hl h)
... | (tri> hl he hg) = ⊥-elim (hl h)

tri-eq-eq : ∀ {A : Set} {a b c : A} (k m : Nat) → (k ≡ m) → (tri k a b c m) ≡ b
tri-eq-eq k m h with (<-cmp k m)
... | (tri< hl he hg) = ⊥-elim (he h)
... | (tri≈ hl he hg) = refl
... | (tri> hl he hg) = ⊥-elim (he h)

tri-eq-gt : ∀ {A : Set} {a b c : A} (k m : Nat) → (k > m) → (tri k a b c m) ≡ c
```

```
tri-eq-gt k m h with (<-cmp k m)
... | (tri< hl he hg) = ⊥-elim (hg h)
... | (tri≈ hl he hg) = ⊥-elim (hg h)
... | (tri> hl he hg) = refl

subst-termoid : {b : Bool} → Nat → Term → Termoid b → Termoid b
subst-termoid k t (var m) = tri k (var (pred m)) t (var m) m
subst-termoid k t (fun f ts) = fun f (subst-termoid k t ts)
subst-termoid k t nil = nil
subst-termoid k t (cons s ts) = cons (subst-termoid k t s) (subst-termoid k t ts)

subst-term : Nat → Term → Term → Term
subst-term k t s = subst-termoid k t s

subst-terms : Nat → Term → Terms → Terms
subst-terms k t ts = subst-termoid k t ts

incr-var : {b : Bool} → Termoid b → Termoid b
incr-var (var k) = var (suc k)
incr-var (fun f ts) = fun f (incr-var ts)
incr-var nil = nil
incr-var (cons t ts) = cons (incr-var t) (incr-var ts)

subst-form : Nat → Term → Formula → Formula
subst-form k t (cst b) = cst b
subst-form k t (not f) = not (subst-form k t f)
subst-form k t (bct b f g) = bct b (subst-form k t f) (subst-form k t g)
subst-form k t (qtf q f) = qtf q (subst-form (suc k) (incr-var t) f)
subst-form k t (rel f ts) = rel f (subst-terms k t ts)

rev-terms : Terms → Terms → Terms
rev-terms nil acc = acc
rev-terms (cons t ts) acc = rev-terms ts (cons t acc)

vars-desc : Nat → Terms
vars-desc 0 = nil
vars-desc (suc k) = cons (var k) (vars-desc k)

vars-asc : Nat → Terms
vars-asc k = rev-terms (vars-desc k) nil

skm-term-asc : Nat → Nat → Term
skm-term-asc k a = fun (nf k) (vars-asc a)

skm-term-desc : Nat → Nat → Term
skm-term-desc k a = fun (nf k) (vars-desc a)

char-to-nat : Char → Maybe Nat
```

```
char-to-nat '0' = just 0
char-to-nat '1' = just 1
char-to-nat '2' = just 2
char-to-nat '3' = just 3
char-to-nat '4' = just 4
char-to-nat '5' = just 5
char-to-nat '6' = just 6
char-to-nat '7' = just 7
char-to-nat '8' = just 8
char-to-nat '9' = just 9
char-to-nat _ = nothing

chars-to-nat-acc : Nat → List Char → Maybe Nat
chars-to-nat-acc k [] = just k
chars-to-nat-acc k (c : cs) = char-to-nat c ?>= \ m → chars-to-nat-acc ((k * 10) + m) cs

chars-to-nat : List Char → Maybe Nat
chars-to-nat = chars-to-nat-acc 0

_⇔_ : Bool → Bool → Bool
true ⇔ true = true
false ⇔ false = true
_ ⇔ _ = false

bct-eq : Bct → Bct → Bool
bct-eq or or = true
bct-eq and and = true
bct-eq imp imp = true
bct-eq iff iff = true
bct-eq _ _ = false

chars-eq : Chars → Chars → Bool
chars-eq [] [] = true
chars-eq (c0 : cs0) (c1 : cs1) = ((c0 =c c1) ∧ (chars-eq cs0 cs1))
chars-eq _ _ = false

ftr-eq : Functor → Functor → Bool
ftr-eq (nf k) (nf m) = k =n m
ftr-eq (sf s') (sf t') = chars-eq s' t'
ftr-eq _ _ = false

termoid-eq : {b1 b2 : Bool} → Termoid b1 → Termoid b2 → Bool
termoid-eq (var k) (var m) = k =n m
termoid-eq (fun f ts) (fun g ss) = ftr-eq f g ∧ termoid-eq ts ss
termoid-eq nil nil = true
termoid-eq (cons t' ts') (cons s' ss') = (termoid-eq t' s') ∧ (termoid-eq ts' ss')
termoid-eq _ _ = false
```

```
eq-term : Term → Term → Bool
eq-term = termoid-eq

terms-eq : Terms → Terms → Bool
terms-eq = termoid-eq

eq-list : {A : Set} → (A → A → Bool) → List A → List A → Bool
eq-list f [] [] = true
eq-list f (x1 : xs1) (x2 : xs2) = f x1 x2 ∧ (eq-list f xs1 xs2)
eq-list f _ _ = false

formula-eq : Formula → Formula → Bool
formula-eq (cst b0) (cst b1) = b0 ⇔ b1
formula-eq (not f) (not g) = formula-eq f g
formula-eq (bct b1 f1 g1) (bct b2 f2 g2) = bct-eq b1 b2 ∧ (formula-eq f1 f2 ∧ formula-eq g1 g2)
formula-eq (qtf p' f') (qtf q' g') = (p' ⇔ q') ∧ (formula-eq f' g')
formula-eq (rel r1 ts1) (rel r2 ts2) = ftr-eq r1 r2 ∧ terms-eq ts1 ts2
formula-eq _ _ = false

pp-digit : Nat → Char
pp-digit 0 = '0'
pp-digit 1 = '1'
pp-digit 2 = '2'
pp-digit 3 = '3'
pp-digit 4 = '4'
pp-digit 5 = '5'
pp-digit 6 = '6'
pp-digit 7 = '7'
pp-digit 8 = '8'
pp-digit 9 = '9'
pp-digit _ = 'E'

{-# NON_TERMINATING #-}
pp-nat : Nat → Chars
pp-nat k = if k <ᵇ 10 then [ pp-digit k ] else (pp-nat (k / 10)) ++ [ (pp-digit (k % 10)) ]

pp-list-core : {A : Set} → (A → String) → List A → String
pp-list-core f [] = "]"
pp-list-core f (x : xs) = concat ("," : f x : pp-list-core f xs : [])

pp-list : {A : Set} → (A → String) → List A → String
pp-list f [] = "[]"
pp-list f (x : xs) = concat ("[" : f x : pp-list-core f xs : [])

pp-ftr : Functor → String
pp-ftr (nf k) = concat ( "#" : show k : [] )
pp-ftr (sf s) = fromList s
```

```
pp-termoid : (b : Bool) → Termoid b → String
pp-termoid false (var k) = concat ( "#" : show k : [] )
pp-termoid false (fun f ts) = concat ( pp-ftr f : "(" : pp-termoid true ts : ")" : [] )
pp-termoid true nil = ""
pp-termoid true (cons t nil) = pp-termoid false t
pp-termoid true (cons t ts) = concat ( pp-termoid false t : "," : pp-termoid true ts : [] )

pp-bool : Bool → String
pp-bool true = "true"
pp-bool false = "false"

pp-term = pp-termoid false
pp-terms = pp-termoid true

pp-form : Formula → String
pp-form (cst true) = "⊤"
pp-form (cst false) = "⊥"
pp-form (rel r ts) = concat ( pp-ftr r : "(" : pp-terms ts : ")" : [] )
pp-form (bct or f g) = concat ( "(" : pp-form f : " ∨ " : pp-form g : ")" : [] )
pp-form (bct and f g) = concat ( "(" : pp-form f : " ∧ " : pp-form g : ")" : [] )
pp-form (bct imp f g) = concat ( "(" : pp-form f : " → " : pp-form g : ")" : [] )
pp-form (bct iff f g) = concat ( "(" : pp-form f : " ↔ " : pp-form g : ")" : [] )
pp-form (qtf true f) = concat ( "∃ " : pp-form f : [] )
pp-form (qtf false f) = concat ( "∀ " : pp-form f : [] )
pp-form (not f) = concat ( "¬ " : pp-form f : [] )

fst : {A : Set} {B : Set} → (A × B) → A
fst (x , _) = x

snd : {A : Set} {B : Set} → (A × B) → B
snd (_ , y) = y

just-if : Bool → Maybe ⊤
just-if true = just tt
just-if false = nothing

suc-inj : ∀ {a b : Nat} → (suc a ≡ suc b) → a ≡ b
suc-inj refl = refl

just-inj : ∀  {A : Set} {a b : A} → (just a ≡ just b) → a ≡ b
just-inj refl = refl

id : ∀ {l} {A : Set l} → A → A
id x = x

elim-eq : ∀ {A : Set} {x : A} {y : A} (p : A → Set) → p x → x ≡ y → p y
elim-eq p h0 refl = h0
```

```
eq-elim : ∀ {A : Set} {x : A} {y : A} (p : A → Set) → x ≡ y → p x → p y
eq-elim p refl = id

eq-elim-symm : ∀ {A : Set} {x : A} {y : A} (p : A → Set) → x ≡ y → p y → p x
eq-elim-symm p refl = id

eq-elim-2 : ∀ {A B : Set} {a0 a1 : A} {b0 b1 : B} (p : A → B → Set) →
  a0 ≡ a1 → b0 ≡ b1 → p a0 b0 → p a1 b1
eq-elim-2 p refl refl = id

eq-elim-3 : ∀ {A B C : Set} {a0 a1 : A} {b0 b1 : B} {c0 c1 : C} (p : A → B → C → Set) →
  a0 ≡ a1 → b0 ≡ b1 → c0 ≡ c1 → p a0 b0 c0 → p a1 b1 c1
eq-elim-3 p refl refl refl = id

eq-elim-4 : ∀ {A B C D : Set} {a0 a1 : A} {b0 b1 : B}
  {c0 c1 : C} {d0 d1 : D} (p : A → B → C → D → Set) →
  a0 ≡ a1 → b0 ≡ b1 → c0 ≡ c1 → d0 ≡ d1 → p a0 b0 c0 d0 → p a1 b1 c1 d1
eq-elim-4 p refl refl refl refl = id

eq-trans : ∀ {A : Set} {x : A} (y : A) {z : A} → x ≡ y → y ≡ z → x ≡ z
eq-trans _ refl refl = refl

eq-symm : ∀ {A : Set} {x : A} {y : A} → x ≡ y → y ≡ x
eq-symm refl = refl

_∈_ : {A : Set} → A → List A → Set
a0 ∈ [] = ⊥
a0 ∈ (a1 ∷ as) = (a0 ≡ a1) ⊎ (a0 ∈ as)

rt : Set → Bool
rt A = elim-lem A (λ _ → true) (λ _ → false)

tr-rt-iff : ∀ {A : Set} → T (rt A) ↔ A
tr-rt-iff {A} with LEM A
... | (yes h0) = (λ _ → h0) , (λ _ → tt)
... | (no h0) = ⊥-elim , h0

F : Bool → Set
F true  = ⊥
F false = ⊤

cong : {A B : Set} (f : A → B) {x y : A} (p : x ≡ y) → f x ≡ f y
cong _ refl = refl

cong-2 : {A B C : Set} (f : A → B → C) {x y : A} {z w : B} (p : x ≡ y) (q : z ≡ w) → f x z ≡ f y w
cong-2 _ refl refl = refl

cong-3 : ∀ {A B C D : Set} (f : A → B → C → D)
```

```
  {a0 a1 : A} {b0 b1 : B} {c0 c1 : C} → a0 ≡ a1 → b0 ≡ b1 → c0 ≡ c1 → f a0 b0 c0 ≡ f a1 b1 c1
cong-3 f refl refl refl = refl

0<s : ∀ k → 0 < suc k
0<s k = s≤s z≤n

s<s⇐< : ∀ k m → (suc k < suc m) → k < m
s<s⇐< k m (s≤s h) = h

ite-intro-lem : ∀ {A : Set} {x y : A} (b : Bool) →
  (P : A → Set) → (T b → P x) → (F b → P y) → P (if b then x else y)
ite-intro-lem false P hx hy = hy tt
ite-intro-lem true P hx hy = hx tt

not-inj₁ : ∀ {A B : Set} → ¬ (A ⊎ B) → ¬ A
not-inj₁ h0 h1 = h0 (inj₁ h1)

not-inj₂ : ∀ {A B : Set} → ¬ (A ⊎ B) → ¬ B
not-inj₂ h0 h1 = h0 (inj₂ h1)

not-imp-lft : ∀ {A B : Set} → ¬ (A → B) → A
not-imp-lft {A} {B} h0 = elim-lem A id \ h1 → ⊥-elim (h0 \ h2 → ⊥-elim (h1 h2))

not-imp-rgt : ∀ {A B : Set} → ¬ (A → B) → ¬ B
not-imp-rgt {A} {B} h0 h1 = ⊥-elim (h0 \ h2 → h1)

imp-to-not-or : ∀ {A B} → (A → B) → ((¬ A) ⊎ B)
imp-to-not-or {A} {B} h0 = elim-lem A (\ h1 → inj₂ (h0 h1)) inj₁

not-and-to-not-or-not : ∀ {A B} → ¬ (A × B) → ((¬ A) ⊎ (¬ B))
not-and-to-not-or-not {A} {B} h0 = elim-lem A
  (\ h1 → elim-lem B (\ h2 → ⊥-elim (h0 (h1 , h2))) inj₂)
  inj₁

prod-inj-lft : ∀ {A B : Set} {a0 a1 : A} {b0 b1 : B} →
  (a0 , b0) ≡ (a1 , b1) → a0 ≡ a1
prod-inj-lft refl = refl

prod-inj-rgt : ∀ {A B : Set} {a0 a1 : A} {b0 b1 : B} →
  (a0 , b0) ≡ (a1 , b1) → b0 ≡ b1
prod-inj-rgt refl = refl

elim-bor : ∀ {A : Set} b1 b2 → (T b1 → A) → (T b2 → A) → T (b1 ∨ b2) → A
elim-bor true _ h0 _ h2 = h0 tt
elim-bor _ true _ h1 h2 = h1 tt

biff-to-eq : ∀ {b0 b1} → T (b0 ⇔ b1) → (b0 ≡ b1)
biff-to-eq {true} {true} _ = refl
biff-to-eq {false} {false} _ = refl
```

```
from-ite : ∀ {A : Set} (P : A → Set) (b : Bool) (a0 a1 : A) →
  P (if b then a0 else a1) → (P a0 ⊎ P a1)
from-ite _ true _ _ = inj₁
from-ite _ false _ _ = inj₂

elim-ite : ∀ {A B : Set} (P : A → Set) (b : Bool) (a0 a1 : A) →
  (P a0 → B) → (P a1 → B) → P (if b then a0 else a1) → B
elim-ite _ true _ _ h0 _ h1 = h0 h1
elim-ite _ false _ _ _ h0 h1 = h0 h1

elim-ite' : ∀ {A B : Set} (P : A → Set) (b : Bool) (a0 a1 : A) →
  P (if b then a0 else a1) → (P a0 → B) → (P a1 → B) → B
elim-ite' P b a0 a1 h h0 h1 = elim-ite P b a0 a1 h0 h1 h

ite-intro : ∀ {A : Set} {x : A} {y : A} (b : Bool) →
  (P : A → Set) → P x → P y → P (if b then x else y)
ite-intro false P hx hy = hy
ite-intro true  P hx hy = hx

iff-to-not-iff-not : ∀ {A B} → (A ↔ B) → ((¬ A) ↔ (¬ B))
iff-to-not-iff-not h0 =
  ( (\ ha hb → ⊥-elim (ha (snd h0 hb))) ,
    (\ hb ha → ⊥-elim (hb (fst h0 ha))) )

or-iff-or : ∀ {A0 A1 B0 B1} → (A0 ↔ A1) → (B0 ↔ B1) → ((A0 ⊎ B0) ↔ (A1 ⊎ B1))
or-iff-or ha hb =
  (\ h0 → or-elim h0
    (\ h1 → (inj₁ (fst ha h1)))
    (\ h1 → (inj₂ (fst hb h1)))) ,
  (\ h0 → or-elim h0
    (\ h1 → (inj₁ (snd ha h1)))
    (\ h1 → (inj₂ (snd hb h1))))

iff-symm : ∀ {A B} → (A ↔ B) → (B ↔ A)
iff-symm h0 = (λ h1 → snd h0 h1) , (λ h1 → fst h0 h1)

iff-trans : ∀ {A} B {C} → (A ↔ B) → (B ↔ C) → (A ↔ C)
iff-trans _ h0 h1 =
  (λ h2 → fst h1 (fst h0 h2)) ,
  (λ h2 → snd h0 (snd h1 h2))

and-iff-and : ∀ {A0 A1 B0 B1} → (A0 ↔ A1) → (B0 ↔ B1) → ((A0 × B0) ↔ (A1 × B1))
and-iff-and ha hb =
  (\ h0 → (fst ha (fst h0) , fst hb (snd h0))) ,
  (\ h0 → (snd ha (fst h0) , snd hb (snd h0)))

imp-iff-imp : ∀ {A0 A1 B0 B1} → (A0 ↔ A1) → (B0 ↔ B1) → ((A0 → B0) ↔ (A1 → B1))
```

```
imp-iff-imp ha hb =
  (\ h0 h1 → fst hb (h0 (snd ha h1))) ,
  (\ h0 h1 → snd hb (h0 (fst ha h1)))

iff-iff-iff : ∀ {A0 A1 B0 B1} → (A0 ↔ A1) → (B0 ↔ B1) → ((A0 ↔ B0) ↔ (A1 ↔ B1))
iff-iff-iff ha hb =
  (λ h0 → iff-trans _ (iff-symm ha) (iff-trans _ h0 hb)) ,
  (λ h0 → iff-trans _ ha (iff-trans _ h0 (iff-symm hb)))

fa-iff-fa : ∀ {A} {P Q : A → Set} → (∀ a → (P a ↔ Q a)) → ((∀ a → P a) ↔ (∀ a → Q a))
fa-iff-fa h0 = ((\ h1 a → fst (h0 a) (h1 a)) , (\h1 a → snd (h0 a) (h1 a)))

ex-iff-ex : ∀ {A} {P Q : A → Set} → (∀ a → (P a ↔ Q a)) → ((∃ P) ↔ (∃ Q))
ex-iff-ex h0 =
  (\ h1 → ex-elim h1 (\ a h2 → a , fst (h0 a) h2)) ,
  (\ h2 → ex-elim h2 (\ a h2 → a , snd (h0 a) h2))

dni : ∀ {A : Set} → A → (¬ (¬ A))
dni h0 h1 = h1 h0

dne : ∀ {A : Set} → (¬ ¬ A) → A
dne {A} h0 = elim-lem A id λ h1 → ⊥-elim (h0 h1)

iff-refl : ∀ {A : Set} → (A ↔ A)
iff-refl = (id , id)

not-iff-not-to-iff : ∀ {A B} → ((¬ A) ↔ (¬ B)) → (A ↔ B)
not-iff-not-to-iff h0 =
  (λ h1 → dne (λ h2 → snd h0 h2 h1)) ,
  (λ h1 → dne (λ h2 → fst h0 h2 h1))

eq-to-iff : ∀ {A : Set} (P : A → Set) (x y : A) → x ≡ y → ((P x) ↔ (P y))
eq-to-iff P x y refl = iff-refl

eq-to-iff-2 : ∀ {A B : Set} (P : A → B → Set) (a0 a1 : A) (b0 b1 : B) →
  a0 ≡ a1 → b0 ≡ b1 → ((P a0 b0) ↔ (P a1 b1))
eq-to-iff-2 P a0 a1 b0 b1 refl refl = iff-refl

bfst : ∀ (a b : Bool) → T (a ∧ b) → T a
bfst true _ _ = tt

bsnd : ∀ a b → T (a ∧ b) → T b
bsnd _ true _ = tt
bsnd true false ()

tr-band-to-and : ∀ a b → T (a ∧ b) → (T a × T b)
tr-band-to-and true true _ = tt , tt

tr-band-to-and-3 : ∀ a b c → T (a ∧ b ∧ c) → (T a × T b × T c)
```

```
tr-band-to-and-3 true true true _ = tt , tt , tt

tr-band-to-and-4 : ∀ a b c d → T (a ∧ b ∧ c ∧ d) → (T a × T b × T c × T d)
tr-band-to-and-4 true true true true _ = tt , tt , tt , tt

tr-band-to-and-5 : ∀ a b c d e → T (a ∧ b ∧ c ∧ d ∧ e) → (T a × T b × T c × T d × T e)
tr-band-to-and-5 true true true true true _ = tt , tt , tt , tt , tt

not-ex-to-fa-not : ∀ {A : Set} (P : A → Set) → (¬ ∃ P) → (∀ x → ¬ P x)
not-ex-to-fa-not P h0 a h1 = h0 (a , h1)

not-fa-to-ex-not : ∀ {A : Set} (P : A → Set) → ¬ (∀ x → P x) → ∃ λ x → ¬ P x
not-fa-to-ex-not P h0 = dne (λ h1 → h0 (λ a → dne (λ h2 → h1 (a , h2))))

not-fst : ∀ {A : Set} {B : Set} → ¬ (A × B) → A → ¬ B
not-fst h0 h1 h2 = h0 (h1 , h2)

tr-to-ite-eq : ∀ {A : Set} {b} {a0 a1 : A} → T b → (if b then a0 else a1) ≡ a0
tr-to-ite-eq {_} {true} _ = refl

fs-to-ite-ne : ∀ {A : Set} {b} {a0 a1 : A} → F b → (if b then a0 else a1) ≡ a1
fs-to-ite-ne {_} {false} _ = refl

char-eq-to-eq : ∀ c0 c1 → T (c0 =c c1) → c0 ≡ c1
char-eq-to-eq c0 c1 = toWitness

chars-eq-to-eq : ∀ cs0 cs1 → T (chars-eq cs0 cs1) → cs0 ≡ cs1
chars-eq-to-eq [] [] _ = refl
chars-eq-to-eq (c0 ∷ cs0) (c1 ∷ cs1) h0 =
  cong-2 _∷_
    (toWitness (bfst (c0 =c c1) _ h0))
    (chars-eq-to-eq cs0 cs1 (bsnd _ _ h0))

ite-elim-lem : ∀ {A B : Set} (P : A → Set) (b : Bool) (a0 a1 : A) →
  (T b → P a0 → B) → (F b → P a1 → B) → P (if b then a0 else a1) → B
ite-elim-lem _ true _ _ h0 _ h1 = h0 tt h1
ite-elim-lem _ false _ _ _ h0 h1 = h0 tt h1

_≠_ : {A : Set} → A → A → Set
x ≠ y = ¬ (x ≡ y)

nf-inj : ∀ {k m} → nf k ≡ nf m → k ≡ m
nf-inj refl = refl

ex-falso : ∀ {A B : Set} → A → ¬ A → B
ex-falso h0 h1 = ⊥-elim (h1 h0)

append-assoc : ∀ {A : Set} (as0 as1 as2 : List A) →
  as0 ++ (as1 ++ as2) ≡ (as0 ++ as1) ++ as2
```

```
append-assoc [] as1 as2 = refl
append-assoc (a : as0) as1 as2 = cong (_:_ a) (append-assoc as0 as1 as2)

reverse-acc-cons : ∀ {A : Set} (as0 as1 : List A) →
  reverseAcc as0 as1 ≡ (reverse as1) ++ as0
reverse-acc-cons as0 [] = refl
reverse-acc-cons as0 (a : as1) =
  eq-trans _ (reverse-acc-cons (a : as0) as1)
    ( eq-trans _ (append-assoc (reverse as1) [ a ] as0)
        ( let h0 : reverse as1 ++ [ a ] ≡ reverseAcc [ a ] as1
              h0 = eq-symm (reverse-acc-cons [ a ] as1) in
          cong (λ x → x ++ as0) h0 ) )

reverse-cons : ∀ {A : Set} (a : A) (as : List A) → reverse (a : as) ≡ (reverse as) :ʳ a
reverse-cons a as = reverse-acc-cons [ a ] as

reverse-app : ∀ {A : Set} (as0 as1 as2 : List A) →
  reverseAcc as0 (as1 ++ as2) ≡ reverseAcc ((reverse as1) ++ as0) as2
reverse-app as0 [] as2 = refl
reverse-app as0 (a : as1) as2 =
  eq-trans _ (reverse-app (a : as0) as1 as2)
    (cong (λ x → reverseAcc x as2)
      (eq-trans _ (append-assoc (reverse as1) [ a ] as0)
        (cong (λ x → x ++ as0) (eq-symm (reverse-cons a as1)))))

app-nil : ∀ {A : Set} (as : List A) → as ++ [] ≡ as
app-nil [] = refl
app-nil (a : as) = cong (_:_ a) (app-nil _)

reverse-snoc : ∀ {A : Set} (a : A) (as : List A) → reverse (as :ʳ a) ≡ a : (reverse as)
reverse-snoc a as = eq-trans _ (reverse-app [] as [ a ]) (cong (_:_ a) (app-nil _))

reverse-reverse : ∀ {A : Set} (as : List A) → reverse (reverse as) ≡ as
reverse-reverse [] = refl
reverse-reverse (a : as) =
  eq-trans (reverse (reverse as :ʳ a))
    (cong reverse (reverse-cons a as))
    ( eq-trans (a : (reverse (reverse as)))
        (reverse-snoc a (reverse as))
        (cong (_:_ a) (reverse-reverse as)) )

intro-elim-lem : ∀ {A B : Set} (C : B → Set) {f : A → B} {g : (¬ A) → B} →
  (∀ (x : A) → C (f x)) → (∀ (x : ¬ A) → C (g x)) → C (elim-lem A f g)
intro-elim-lem {A} {B} C {f} {g} hf hg with LEM A
... | (yes h0) = hf h0
... | (no h0) = hg h0
```

```
intro-elim-lem-yes : ∀ {A B : Set} (C : B → Set) {f : A → B} {g : (¬ A) → B} →
  (∀ (x : A) → C (f x)) → A → C (elim-lem A f g)
intro-elim-lem-yes {A} {B} C {f} {g} hf hA = intro-elim-lem C hf λ h0 → ⊥-elim (h0 hA)

not-app-eq-nil : ∀ {A : Set} (a : A) as0 as1 → (as0 ++ (a : as1)) ≢ []
not-app-eq-nil _ [] _ ()
not-app-eq-nil _ (_ : _) _ ()

cons-inj : ∀ {A : Set} (a0 a1 : A) as0 as1 → a0 : as0 ≡ a1 : as1 → (a0 ≡ a1) × (as0 ≡ as1)
cons-inj a0 a1 as0 as1 refl = refl , refl

snoc-inj : ∀ {A : Set} (a0 a1 : A) as0 as1 → as0 :ʳ a0 ≡ as1 :ʳ a1 → (as0 ≡ as1) × (a0 ≡ a1)
snoc-inj a0 a1 [] [] refl = refl , refl
snoc-inj a0 a1 (a0' : as0) [] h0 = ⊥-elim (not-app-eq-nil _ _ _ (snd (cons-inj a0' a1 _ _ h0)))
snoc-inj a0 a1 [] (a1' : as1) h0 = ⊥-elim (not-app-eq-nil _ _ _ (snd (cons-inj a1' a0 _ _ (eq-symm h0)))
snoc-inj a0 a1 (a0' : as0) (a1' : as1) h0 =
  let (h1 , h2) = cons-inj a0' a1' _ _ h0 in
  let (h3 , h4) = snoc-inj a0 a1 as0 as1 h2 in
  cong-2 _:_ h1 h3 , h4

reverse-inj : ∀ {A : Set} (as0 as1 : List A) → reverse as0 ≡ reverse as1 → as0 ≡ as1
reverse-inj [] [] refl = refl
reverse-inj (a0 : as0) [] h0 = ⊥-elim (not-app-eq-nil _ _ _ (eq-trans _ (eq-symm (reverse-cons a0 as0)) h
reverse-inj [] (a1 : as1) h0 = ⊥-elim (not-app-eq-nil _ _ _ (eq-symm (eq-trans _ h0 ( (reverse-cons a1 as
reverse-inj (a0 : as0) (a1 : as1) h0 =
  let h3 = eq-symm (reverse-cons a0 as0) in
  let h4 = reverse-cons a1 as1 in
  let (h1 , h2) = snoc-inj a0 a1 (reverse as0) (reverse as1) (eq-trans _ h3 (eq-trans _ h0 h4)) in
  cong-2 _:_ h2 (reverse-inj _ _ h1)

cong-fun-arg : ∀ {A B : Set} {x0 x1 : A → B} {y0 y1 : A} →
  x0 ≡ x1 → y0 ≡ y1 → (x0 y0 ≡ x1 y1)
cong-fun-arg refl refl = refl

elim-eq-symm : ∀ {A : Set} {x : A} {y : A} (p : A → Set) → p y → x ≡ y → p x
elim-eq-symm p h0 refl = h0

data Tree (A : Set) : Set where
  nil : Tree A
  fork : Nat → A → Tree A → Tree A → Tree A

size : {A : Set} → Tree A → Nat
size nil = 0
size (fork k _ _ _) = k

add : {A : Set} → Tree A → A → Tree A
add nil a = fork 1 a nil nil
add ts@(fork k b t s) a =
```

```
  if (size s <ᵇ size t)
  then (fork (k + 1) b t (add s a))
  else (fork (k + 1) a ts nil)


add-fork-intro : {A : Set} (r : Tree A → Set) (k : Nat) (a b : A) (t s : Tree A) →
  (r (fork (k + 1) a t (add s b))) →
  (r (fork (k + 1) b (fork k a t s) nil)) →
  r (add (fork k a t s) b)
add-fork-intro r k a b t s h0 h1 = ite-intro (size s <ᵇ size t) r h0 h1

from-add-fork : {A : Set} (r : Tree A → Set) (k : Nat) (t s : Tree A) (a b : A) →
  r (add (fork k b t s) a) →
  (r (fork (k + 1) b t (add s a)) ⊎ r (fork (k + 1) a (fork k b t s) nil))
from-add-fork r k t s a b = from-ite r (size s <ᵇ size t) _ _

mem : {A : Set} → A → Tree A → Set
mem _ nil = ⊥
mem a (fork _ b t s) = mem a t ⊎ (a ≡ b) ⊎ mem a s

from-mem-add : {A : Set} (t : Tree A) (a b : A) → mem a (add t b) → (mem a t ⊎ (a ≡ b))
from-mem-add nil a b = or-elim' ⊥-elim (or-elim' inj₂ ⊥-elim)
from-mem-add (fork k c t s) a b h0 =
  or-elim (from-add-fork (λ x → mem a x) k t s b c h0)
    ( or-elim' (intro-or-lft inj₁)
      ( or-elim' (intro-or-lft (intro-or-rgt inj₁))
        λ h1 →
          or-elim (from-mem-add s a b h1)
            (intro-or-lft (intro-or-rgt inj₂)) inj₂ ) )
    (or-elim' inj₁ (or-elim' inj₂ ⊥-elim))

all : {A : Set} (p : A → Set) (t : Tree A) → Set
all p t = ∀ a → mem a t → p a

all-sub-lft : {A : Set} (p : A → Set) (k : Nat) (t s : Tree A) (a : A) →
  all p (fork k a t s) → all p t
all-sub-lft p k t s b h0 a h1 = h0 a (inj₁ h1)

all-sub-rgt : {A : Set} (p : A → Set) (k : Nat) (t s : Tree A) (a : A) →
  all p (fork k a t s) → all p s
all-sub-rgt p k t s b h0 a h1 = h0 a (inj₂ (inj₂ h1))

all-add : {A : Set} (p : A → Set) (t : Tree A) (a : A) →
  all p t → p a → all p (add t a)
all-add p t a h0 h1 c h2 = or-elim (from-mem-add t c a h2) (h0 c) (elim-eq-symm p h1)

size-add : {A : Set} (t : Tree A) (a : A) → size (add t a) ≡ suc (size t)
size-add nil a = refl
```

```
size-add (fork k b t0 t1) a =
  add-fork-intro (λ x → size x ≡ suc k) k b a t0 t1 (+-comm k 1) (+-comm k 1)

lookup : {A : Set} → Nat → Tree A → A → A
lookup _ nil a = a
lookup k (fork _ b t s) a =
  tri k (lookup k t a) b (lookup (k - (size t + 1)) s a) (size t)

mem-lookup : {A : Set} (t : Tree A) (k : Nat) (a : A) →
  mem (lookup k t a) t ⊎ (lookup k t a ≡ a)
mem-lookup nil _ _ = inj₂ refl
mem-lookup t@(fork m b t0 t1) k a =
  tri-intro-lem k (size t0) (λ x → (mem x t ⊎ (x ≡ a)))
    (λ _ → or-elim (mem-lookup t0 k a) (intro-or-lft inj₁) inj₂)
    (λ _ → inj₁ (inj₂ (inj₁ refl)))
    ( λ _ → or-elim (mem-lookup t1 (k - (size t0 + 1)) a)
      (intro-or-lft (intro-or-rgt inj₂))
      inj₂ )

pred-suc-eq-suc-pred : ∀ k → 0 < k → pred (suc k) ≡ suc (pred k)
pred-suc-eq-suc-pred (suc k) h0 = refl

n<sn : ∀ n → n < suc n
n<sn n = ≤-refl {suc n}
```