Chapter 1

Library TilingProgram

1.1 Preference

Require Import *Ssreflect.ssreflect Ssreflect.ssrnat Ssreflect.ssrbool Ssreflect.ssrfun Ssreflect.eqtype*.
Require Import *ExtrOcamlNatInt ExtrOcamlString*.

!=を,

Notation "n '!=' m" := $((n == m) = \mathit{false})$.

Lemma *eq_or_neq* : $\forall$ $(n\ m : \mathit{nat})$, $(n == m) \lor (n\ != m)$.
Proof.
move $\Rightarrow$ $n\ m$.
induction $(n == m)$; simpl.
by [left].
by [right].
Qed.

特定の $C0$ 以外なら何でもいい場合は, C_other2 C0 で他の色を求める.

Definition *C_other2* (*C0* : *nat*) : *nat* :=
match *C0* with
  | 0 $\Rightarrow$ 1
  | _ $\Rightarrow$ 0
end.

Lemma *C_other2_neq* : $\forall$ *C0* : *nat*, *C0* != *C_other2 C0*.
Proof.
case $\Rightarrow$ [|]//.
Qed.

Lemma *C_other2_neq'* : $\forall$ *C0* : *nat*, *C_other2 C0* != *C0*.
Proof.
case $\Rightarrow$ [|]//.
Qed.

3 色以上使える環境で, 色 $C0$ と $C1$ が指定されたとき, それらと異なる色を返す関数.

```
Definition C_other3 (C0 C1 : nat) : nat :=
match C0 with
  | 0 ⇒ match C1 with
          | 0 ⇒ 1
          | 1 ⇒ 2
          | _ ⇒ 1
        end
  | 1 ⇒ match C1 with
          | 0 ⇒ 2
          | _ ⇒ 0
        end
  | _ ⇒ match C1 with
          | 0 ⇒ 1
          | _ ⇒ 0
        end
end.
```

```
Lemma C_other3_neq :
 ∀ (C0 C1 : nat), C0 != C_other3 C0 C1 ∧ C1 != C_other3 C0 C1.
Proof.
move ⇒ C0 C1.
induction C0.
induction C1.
by [compute].
induction C1.
by [compute].
by [compute].
induction C0.
induction C1.
by [compute].
by [compute].
induction C1.
by [compute].
by [compute].
Qed.
```

```
Lemma C_other3_neq1 :
 ∀ (C0 C1 : nat), C0 != C_other3 C0 C1.
Proof.
apply C_other3_neq.
Qed.
```

```
Lemma C_other3_neq2 :
```

```
 ∀ (C0 C1 : nat), C1 != C_other3 C0 C1.
Proof.
apply C_other3_neq.
Qed.
Lemma C_other3_neq1' :
 ∀ (C0 C1 : nat), C_other3 C0 C1 != C0.
Proof.
move ⇒ C0 C1.
rewrite eq_sym.
apply C_other3_neq.
Qed.
Lemma C_other3_neq2' :
 ∀ (C0 C1 : nat), C_other3 C0 C1 != C1.
Proof.
move ⇒ C0 C1.
rewrite eq_sym.
apply C_other3_neq.
Qed.
```

1.2 Wang tiling

境界条件とエッジ関数は, ともに“x 座標と y 座標から色を返す関数”である.

```
Definition boundary := nat → nat → nat.
Definition edge := nat → nat → nat.
```

テスト用にプログラムを用いたタイリングを表示する関数も作ってみる.

```
Definition null {A : Type} (x : A): A.
Proof.
apply x.
Qed.
Notation "'^'" := (null 0).
Notation "'#'" := (null 1).
Open Scope list_scope.
Fixpoint e_i (j : nat) : edge → nat → list nat :=
 fun (e : edge)(i : nat) ⇒
 match j with
   | 0 ⇒ ^ :: nil
   | S j' ⇒ (e_i j' e i) ++ ((e i (S j')) :: ^ :: nil)
 end.
Fixpoint e'_i (j : nat) : edge → nat → list nat :=
 fun (e' : edge)(i : nat) ⇒
 match j with
```

```
    | 0 ⇒ (e' i 0) :: nil
    | S j' ⇒ (e'_i j' e' i) ++ (# :: (e' i (S j')) :: nil)
  end.
Fixpoint e_e' (n m : nat)(e e' : edge) : list (list nat) :=
  match n with
    | 0 ⇒ (e_i m e 0) :: nil
    | S n' ⇒ (e_e' n' m e e') ++ ((e'_i m e' (S n')) :: (e_i m e (S n')) :: nil)
  end.
Definition tiling (n m : nat)(b : boundary)(e_ e'_ : boundary → edge) := e_e' n m (e_ b) (e'_ b).
```

長方形サイズ $n \times m$, 境界条件 `b`, エッジ関数 `e_`, `e'_`から実際のタイリングを求める関数. まずは P_{12} をタイリングする関数から. `e` は横エッジ用, `e'` は縦エッジ用.

```
Definition e_12 (b : boundary) : edge.
```

横エッジはそのまま, `e 0 j = b 0 j`, `e 1 j = b 2 j` とすればよい.

```
rewrite /edge.
apply (fun i j : nat ⇒
match i with
  | 0 ⇒ b 0 j
  | _ ⇒ b 2 j
end).
Defined.
```

```
Definition e'_12 (b : boundary) : edge.
```

`e' 1 0 = b 1 0`, `e' 1 2 = b 1 3` なので, j で induction.

```
rewrite /edge.
apply (fun i j : nat ⇒
match j with
  | 0 ⇒ b 1 0
  | 1 ⇒ if b 1 0 == b 1 3
          then
           (if b 0 1 == b 2 1 then C_other2 (b 1 0) else b 1 0)
          else
           (if b 0 1 == b 2 1
             then
              (if b 0 2 == b 2 2 then C_other3 (b 1 0) (b 1 3) else b 1 3)
             else b 1 0)
  | _ ⇒ b 1 3
end).
Defined.
```

次に P_{22} をタイリングする関数.

```
Definition e_22 (b : boundary) : edge.
rewrite /edge.
```

```
apply (fun i j : nat ⇒
match i with
  | 0 ⇒ b 0 j
  | 1 ⇒ if b 1 0 == b 1 3
          then
           (if b 2 0 == b 2 3
             then C_other3 (b 0 j) (b 3 j)
             else
              (if b 0 1 == b 3 1
                then b 0 j
                else
                 (if b 0 2 == b 3 2
                   then (match j with
                           | 0 | 1 ⇒ b 3 1
                           | _ ⇒ C_other2 (b 0 j)
                         end)
                   else b 3 j)))
          else
           (if b 2 0 == b 2 3
             then
              (if b 0 1 == b 3 1
                then b 3 j
                else
                 (if b 0 2 == b 3 2
                   then (match j with
                           | 0 | 1 ⇒ b 0 1
                           | _ ⇒ C_other2 (b 3 j)
                        end)
                   else b 0 j))
             else (match j with
                     | 0 | 1 ⇒ b 0 1
                     | _ ⇒ b 3 2
                   end))
  | _ ⇒ b 3 j
end).
Defined.
```

Definition *e'_22* (*b* : *boundary*) : *edge*.

上で定義した e_22 に基づいて定義する.

```
rewrite /edge.
apply (fun i j : nat ⇒
match i with
  | 0 ⇒ 0
```

```
| 1 ⇒ (e'_12 (fun i j : nat ⇒
                match i with
                  | 0 ⇒ b 0 j
                  | 1 ⇒ b 1 j
                  | _ ⇒ (e_22  b 1 j)
                end) 1 j)
| _ ⇒ (e'_12 (fun i j : nat ⇒
                match i with
                  | 0 ⇒ (e_22  b 1 j)
                  | _ ⇒ b (S  i) j
                end) 1 j)
end
).
Defined.
```

$P_{(n+1)m}$ の境界条件を, P_{nm} と P_{1m} に分割し, 前者の境界条件を出す関数.

```
Definition bSnm_to_bnm (m : nat) : boundary → boundary.
move ⇒ b.
rewrite /boundary.
apply (fun i j : nat ⇒
match m with
  | 0 ⇒ b i j
  | 1 ⇒ b i j
  | _ ⇒ match i with
          | 0 ⇒ match j with
                  | 0 ⇒ 0
                  | 1 ⇒ b 0 1
                  | 2 ⇒ b 0 2
                  | _ ⇒ C_other2 (b 0 j)
                 end
          | _ ⇒ b (S  i) j
        end
end).
Defined.
```

$P_{(n+1)m}$ の境界条件を, P_{nm} と P_{1m} に分割し, 後者の境界条件を出す関数.

```
Definition bSnm_to_b1m (m : nat) : boundary → boundary.
move ⇒ b.
rewrite /boundary.
apply (fun i j : nat ⇒
match m with
  | 0 ⇒ b i j
  | 1 ⇒ b i j
```

```
  | _ ⇒ match i with
            | 0 ⇒ b 0 j
            | 1 ⇒ b 1 j
            | _ ⇒ match j with
                      | 0 ⇒ 0
                      | 1 ⇒ b 0 1
                      | 2 ⇒ b 0 2
                      | _ ⇒ C_other2 (b 0 j)
                    end
          end
end).
Defined.
```

bSnm_to_b1m で出てくる P_{1m} をタイリングする関数.

```
Definition e_1m (b : boundary) : edge.
```

横エッジはそのまま, `e 0 j = b 0 j`, `e 1 j = b 2 j` とすればよい

```
rewrite /edge.
apply (fun i j : nat ⇒
match i with
  | 0 ⇒ b 0 j
  | _ ⇒ b 2 j
end).
Defined.
```

```
Definition e'_1m (m : nat)(b : boundary) : edge.
```

縦エッジは, `b 1 0 = e' 1 0 <> e' 1 1 <> e' 1 2 = ... = e' 1 m = b 1 (S m)` にする

```
rewrite /edge.
apply (fun i j : nat ⇒
match i with
  | 0 ⇒ 0
  | _ ⇒ match j with
            | 0 ⇒ b 1 0
            | 1 ⇒ C_other3 (b 1 0) (b 1 (S m))
            | _ ⇒ b 1 (S m)
          end
end).
Defined.
```

P_{n2} を Tiling する関数.

```
Fixpoint e_n2 (n : nat) : boundary → edge :=
 fun b : boundary ⇒
 match n with
   | 0 | 1 ⇒ e_12 b
```

```
   | 2 ⇒ e_22 b
   | S n' ⇒ fun (i j : nat) ⇒
            match i with
              | 0 ⇒ (bSnm_to_b1m 2 b) 0 j
              | S i' ⇒ e_n2 n' (bSnm_to_bnm 2 b) i' j
            end
 end.
```

```
Fixpoint e'_n2 (n : nat) : boundary → edge :=
 fun b : boundary ⇒
 match n with
   | 0 | 1 ⇒ e'_12 b
   | 2 ⇒ e'_22 b
   | S n' ⇒ fun (i j : nat) ⇒
            match i with
              | 0 ⇒ 0
              | 1 ⇒ e'_1m 2 (bSnm_to_b1m 2 b) 1 j
              | S i' ⇒ e'_n2 n' (bSnm_to_bnm 2 b) i' j
            end
 end.
```

P_{nm} での境界条件および Tiling 関数を P_{mn} のものに置き換える関数. やっていることはただの引数順序の入れ替え. 横エッジ e と縦エッジ e' も入れ替える.

```
Definition bnm_to_bmn (b : boundary) : boundary.
move ⇒ i j.
apply (b j i).
Defined.
```

```
Definition enm_to_emn (e : boundary → edge) : boundary → edge.
move ⇒ b i j.
apply (e (bnm_to_bmn b) j i).
Defined.
```

3 色以上, 2×2 以上のときに, P_{nm} を Tiling する関数.

```
Fixpoint e_nm (n m : nat) : boundary → edge :=
 fun b : boundary ⇒
 match n with
   | 0 | 1 ⇒ e_1m b
   | 2 ⇒ enm_to_emn (fun b' ⇒ e'_n2 m b') b
   | S n' ⇒ fun (i j : nat) ⇒
            match i with
              | 0 ⇒ (bSnm_to_b1m m b) 0 j
              | S i' ⇒ e_nm n' m (bSnm_to_bnm m b) i' j
            end
 end.
```

```
Fixpoint e'_nm (n m : nat) : boundary → edge :=
 fun b : boundary ⇒
 match n with
   | 0 | 1 ⇒ e'_1m m b
   | 2 ⇒ enm_to_emn (fun b' ⇒ e_n2 m b') b
   | S n' ⇒ fun (i j : nat) ⇒
               match i with
                 | 0 ⇒ 0
                 | 1 ⇒ e'_1m m (bSnm_to_b1m m b) 1 j
                 | S i' ⇒ e'_nm n' m (bSnm_to_bnm m b) i' j
               end
 end.
```

Tiling 関数を e_nm, e'_nm に固定したものを定義.

```
Definition tiling_nm (n m : nat)(b : boundary) :=
 tiling n m b (e_nm n m) (e'_nm n m).
```

1.3 Examples

```
Compute (C_other2 1).

     = 0
     : nat

Compute (C_other3 0 1).

     = 2
     : nat

Compute (C_other3 2 0).

     = 1
     : nat

Compute (tiling 1 2 (fun _ _ ⇒ 0) e_12 e'_12).

     = (^ :: 0 :: ^ :: 0 :: ^ :: nil)
    :: (0 :: # :: 1 :: # :: 0 :: nil)
    :: (^ :: 0 :: ^ :: 0 :: ^ :: nil) :: nil
     : list (list nat)
```

がタイルを表す. つまり # の上下左右が Brick Corner Wang Tiling の条件を満たせばよい.

```
Compute (tiling 1 2 (fun _ j ⇒ match j with 1 ⇒ 2 | _ ⇒ 1 end) e_12 e'_12).
```

```
     = (^ :: 2 :: ^ :: 1 :: ^ :: nil)
    :: (1 :: # :: 0 :: # :: 1 :: nil)
    :: (^ :: 2 :: ^ :: 1 :: ^ :: nil) :: nil
     : list (list nat)
```

`Definition` *boundary22a* (i j:*nat*) := 0.

`Compute` (*tiling* 2 2 *boundary22a* *e_22* *e'_22*).

```
     = (^ :: 0 :: ^ :: 0 :: ^ :: nil)
    :: (0 :: # :: 0 :: # :: 0 :: nil)
    :: (^ :: 1 :: ^ :: 1 :: ^ :: nil)
    :: (0 :: # :: 0 :: # :: 0 :: nil)
    :: (^ :: 0 :: ^ :: 0 :: ^ :: nil) :: nil
     : list (list nat)
```

`Definition` *boundary22b* (i j:*nat*) :=
  `match` i `with` 0 ⇒ 2 | _ ⇒ `match` j `with` 0 ⇒ 0 | _ ⇒ 1 `end` `end`.

`Compute` (*tiling* 2 2 *boundary22b* *e_22* *e'_22*).

```
     = (^ :: 2 :: ^ :: 2 :: ^ :: nil)
    :: (0 :: # :: 1 :: # :: 1 :: nil)
    :: (^ :: 2 :: ^ :: 1 :: ^ :: nil)
    :: (0 :: # :: 0 :: # :: 1 :: nil)
    :: (^ :: 1 :: ^ :: 1 :: ^ :: nil) :: nil
     : list (list nat)
```

`Definition` *boundary22c* (i j:*nat*) :=
  `match` j `with` 1 ⇒ 2 | 3 ⇒ 1 | _ ⇒ `match` i `with` 1 ⇒ 0 | _ ⇒ 1 `end` `end`.

`Compute` (*tiling* 2 2 *boundary22c* *e_22* *e'_22*).

```
     = (^ :: 2 :: ^ :: 1 :: ^ :: nil)
    :: (0 :: # :: 2 :: # :: 1 :: nil)
    :: (^ :: 2 :: ^ :: 1 :: ^ :: nil)
    :: (1 :: # :: 0 :: # :: 1 :: nil)
    :: (^ :: 2 :: ^ :: 1 :: ^ :: nil) :: nil
     : list (list nat)
```

`Compute` (*tiling* 1 4 (*bSnm_to_b1m* 4 (`fun` i j ⇒ `match` i `with` 0 ⇒ 2 | _ ⇒ `match` j `with` 0 ⇒ 0 | _ ⇒ 1 `end` `end`)) *e_1m* (*e'_1m* 4)).

```
     = (^ :: 2 :: ^ :: 2 :: ^ :: 2 :: ^ :: 2 :: ^ :: nil)
    :: (0 :: # :: 2 :: # :: 1 :: # :: 1 :: # :: 1 :: nil)
    :: (^ :: 2 :: ^ :: 2 :: ^ :: 0 :: ^ :: 0 :: ^ :: nil) :: nil
     : list (list nat)
```

Compute (*tiling* 1 4 (*bSnm_to_b1m* 4 (fun *i j* ⇒ match *j* with 1 ⇒ 2 | 3 ⇒ 1 | _ ⇒ match *i* with 1 ⇒ 0 | _ ⇒ 1 end end)) *e_1m* (*e'_1m* 4)).

```
 = (^ :: 2 :: ^ :: 1 :: ^ :: 1 :: ^ :: 1 :: ^ :: nil)
:: (0 :: # :: 1 :: # :: 0 :: # :: 0 :: # :: 0 :: nil)
:: (^ :: 2 :: ^ :: 1 :: ^ :: 0 :: ^ :: 0 :: ^ :: nil) :: nil
 : list (list nat)
```

Compute (*tiling* 3 2 (fun *i j* ⇒ match *i* with 0 ⇒ 2 | _ ⇒ match *j* with 0 ⇒ 0 | _ ⇒ 1 end end) (*e_n2* 3) (*e'_n2* 3)).

```
 = (^ :: 2 :: ^ :: 2 :: ^ :: nil)
:: (0 :: # :: 2 :: # :: 1 :: nil)
:: (^ :: 2 :: ^ :: 2 :: ^ :: nil)
:: (0 :: # :: 1 :: # :: 1 :: nil)
:: (^ :: 2 :: ^ :: 1 :: ^ :: nil)
:: (0 :: # :: 0 :: # :: 1 :: nil)
:: (^ :: 1 :: ^ :: 1 :: ^ :: nil) :: nil
 : list (list nat)
```

Compute (*tiling* 4 2 (fun *i j* ⇒ match *j* with 1 ⇒ 2 | 3 ⇒ 1 | _ ⇒ match *i* with 1 ⇒ 0 | _ ⇒ 1 end end) (*e_n2* 4) (*e'_n2* 4)).

```
 = (^ :: 2 :: ^ :: 1 :: ^ :: nil)
:: (0 :: # :: 2 :: # :: 1 :: nil)
:: (^ :: 2 :: ^ :: 1 :: ^ :: nil)
:: (1 :: # :: 0 :: # :: 1 :: nil)
:: (^ :: 2 :: ^ :: 1 :: ^ :: nil)
:: (1 :: # :: 1 :: # :: 1 :: nil)
:: (^ :: 0 :: ^ :: 0 :: ^ :: nil)
:: (1 :: # :: 1 :: # :: 1 :: nil)
:: (^ :: 2 :: ^ :: 1 :: ^ :: nil) :: nil
 : list (list nat)
```

Compute (*tiling* 2 4 (*bnm_to_bmn* (fun *i j* ⇒ match *i* with 0 ⇒ 2 | _ ⇒ match *j* with 0 ⇒ 0 | _ ⇒ 1 end end)) (*enm_to_emn* (*e'_n2* 4)) (*enm_to_emn* (*e_n2* 4))).

```
 = (^ :: 0 :: ^ :: 0 :: ^ :: 0 :: ^ :: 0 :: ^ :: nil)
:: (2 :: # :: 2 :: # :: 2 :: # :: 2 :: # :: 1 :: nil)
:: (^ :: 2 :: ^ :: 2 :: ^ :: 1 :: ^ :: 0 :: ^ :: nil)
:: (2 :: # :: 2 :: # :: 2 :: # :: 1 :: # :: 1 :: nil)
:: (^ :: 1 :: ^ :: 1 :: ^ :: 1 :: ^ :: 1 :: ^ :: nil) :: nil
 : list (list nat)
```

Compute (*tiling* 2 4 (*bnm_to_bmn* (fun *i j* ⇒ match *j* with 1 ⇒ 2 | 3 ⇒ 1 | _ ⇒ match *i* with 1 ⇒ 0 | _ ⇒ 1 end end)) (*enm_to_emn* (*e'_n2* 4)) (*enm_to_emn* (*e_n2* 4))).

```
 = (^ :: 0 :: ^ :: 1 :: ^ :: 1 :: ^ :: 1 :: ^ :: nil)
:: (2 :: # :: 2 :: # :: 2 :: # :: 0 :: # :: 2 :: nil)
:: (^ :: 2 :: ^ :: 0 :: ^ :: 1 :: ^ :: 1 :: ^ :: nil)
:: (1 :: # :: 1 :: # :: 1 :: # :: 0 :: # :: 1 :: nil)
:: (^ :: 1 :: ^ :: 1 :: ^ :: 1 :: ^ :: 1 :: ^ :: nil) :: nil
 : list (list nat)
```

Definition *boundary44a* (*i j*:*nat*) :=
match *i* with 0 ⇒ 2 | 3 ⇒ match *j* with 0 ⇒ 5 | _ ⇒ 1 end | _ ⇒ match *j* with 1 ⇒ 3 | _ ⇒ 4 end end.

Compute (*tiling_nm* 4 4 *boundary44a*).

```
 = (^ :: 2 :: ^ :: 2 :: ^ :: 2 :: ^ :: 2 :: ^ :: nil)
:: (4 :: # :: 0 :: # :: 4 :: # :: 4 :: # :: 4 :: nil)
:: (^ :: 2 :: ^ :: 2 :: ^ :: 0 :: ^ :: 0 :: ^ :: nil)
:: (4 :: # :: 0 :: # :: 4 :: # :: 4 :: # :: 4 :: nil)
:: (^ :: 2 :: ^ :: 2 :: ^ :: 1 :: ^ :: 1 :: ^ :: nil)
:: (5 :: # :: 5 :: # :: 5 :: # :: 5 :: # :: 1 :: nil)
:: (^ :: 0 :: ^ :: 0 :: ^ :: 0 :: ^ :: 1 :: ^ :: nil)
:: (4 :: # :: 4 :: # :: 4 :: # :: 4 :: # :: 4 :: nil)
:: (^ :: 3 :: ^ :: 4 :: ^ :: 4 :: ^ :: 4 :: ^ :: nil) :: nil
 : list (list nat)
```

Definition *boundary44b* (*i j*:*nat*) :=
match *j* with 0 ⇒ match *i* with 2 | 3 ⇒ 3 | _ ⇒ 4 end | 1 ⇒ 2 | 3 ⇒ 1 | _ ⇒ match *i* with 0 ⇒ 0 | _ ⇒ 5 end end.

Compute (*tiling_nm* 4 4 *boundary44b*).

```
 = (^ :: 2 :: ^ :: 0 :: ^ :: 1 :: ^ :: 0 :: ^ :: nil)
:: (4 :: # :: 0 :: # :: 5 :: # :: 5 :: # :: 5 :: nil)
:: (^ :: 2 :: ^ :: 0 :: ^ :: 0 :: ^ :: 1 :: ^ :: nil)
:: (3 :: # :: 0 :: # :: 5 :: # :: 5 :: # :: 5 :: nil)
:: (^ :: 2 :: ^ :: 0 :: ^ :: 1 :: ^ :: 0 :: ^ :: nil)
:: (3 :: # :: 3 :: # :: 3 :: # :: 5 :: # :: 5 :: nil)
:: (^ :: 0 :: ^ :: 1 :: ^ :: 1 :: ^ :: 1 :: ^ :: nil)
:: (4 :: # :: 4 :: # :: 4 :: # :: 5 :: # :: 5 :: nil)
:: (^ :: 2 :: ^ :: 5 :: ^ :: 1 :: ^ :: 5 :: ^ :: nil) :: nil
 : list (list nat)
```

Definition *boundary44c* (i j:*nat*) :=
  match j with $0 \Rightarrow$ match i with $2 \mid 3 \Rightarrow 3 \mid \_ \Rightarrow 2$ end $\mid 1 \Rightarrow 2 \mid 3 \Rightarrow 1 \mid \_ \Rightarrow$ match i with $0 \Rightarrow 0 \mid \_ \Rightarrow 1$ end end.

Compute (*tiling_nm* 4 4 *boundary44c*).

```
 = (^ :: 2 :: ^ :: 0 :: ^ :: 1 :: ^ :: 0 :: ^ :: nil)
:: (2 :: # :: 0 :: # :: 1 :: # :: 1 :: # :: 1 :: nil)
:: (^ :: 2 :: ^ :: 0 :: ^ :: 0 :: ^ :: 1 :: ^ :: nil)
:: (3 :: # :: 0 :: # :: 1 :: # :: 1 :: # :: 1 :: nil)
:: (^ :: 2 :: ^ :: 0 :: ^ :: 1 :: ^ :: 0 :: ^ :: nil)
:: (3 :: # :: 3 :: # :: 3 :: # :: 1 :: # :: 1 :: nil)
:: (^ :: 0 :: ^ :: 2 :: ^ :: 1 :: ^ :: 2 :: ^ :: nil)
:: (2 :: # :: 2 :: # :: 2 :: # :: 1 :: # :: 1 :: nil)
:: (^ :: 2 :: ^ :: 1 :: ^ :: 1 :: ^ :: 1 :: ^ :: nil) :: nil
 : list (list nat)
```

1.4 Main theorems

WangTiling.v から, “Tiling 可能” の定義をインポート．以下の 3 つを全て同時に満たせば Tiling できていることになる．

Definition *Boundary_i* (n m : *nat*)(b : *boundary*)(e' : *edge*) :=
 $\forall$ i : *nat*, e' i 0 == b i 0 $\land$ e' i m == b i (S m) $\lor$ $i = 0 \lor n < i$.
Definition *Boundary_j* (n m : *nat*)(b : *boundary*)(e : *edge*) :=
 $\forall$ j : *nat*, e 0 j == b 0 j $\land$ e n j == b (S n) j $\lor$ $j = 0 \lor m < j$.
Definition *Brick* (n m : *nat*)(e e' : *edge*) :=
 $\forall$ i j : *nat*,
 (e i (S j) == e (S i) (S j) $\land$ e' (S i) j != e' (S i) (S j)) $\lor$
 (e i (S j) != e (S i) (S j) $\land$ e' (S i) j == e' (S i) (S j)) $\lor$
 $n \le i \lor m \le j$.
Definition *Valid* (n m : *nat*)(b : *boundary*)(e e' : *edge*) :=
 Boundary_i n m b e' $\land$ *Boundary_j* n m b e $\land$ *Brick* n m e e'.
Definition *Valid_nm* (n m : *nat*)(b : *boundary*) :=
 Boundary_i n m b (*e'_nm* n m b) $\land$ *Boundary_j* n m b (*e_nm* n m b) $\land$
 Brick n m (*e_nm* n m b) (*e'_nm* n m b).
Ltac *e_unfold* := rewrite /*e_nm*/*e'_nm*/*enm_to_emn*/*e_n2*/*e'_n2*/*bnm_to_bmn*/*e'_22*/*e_22*/*e'_12*.
Ltac *eq_rewrite* :=
 repeat match goal with
   | [H : *is_true* (_ == _) $\vdash$ _] $\Rightarrow$ rewrite (*elimTF* *eqP* H)
   | [H : (_ != _) $\vdash$ _] $\Rightarrow$ rewrite H
  end;
 try repeat rewrite *eq_refl*.
Ltac *C_other* :=

```
 repeat match goal with
        | [_ : _ ⊢ _] ⇒ rewrite C_other2_neq
        | [_ : _ ⊢ _] ⇒ rewrite C_other2_neq'
        | [_ : _ ⊢ _] ⇒ rewrite C_other3_neq1
        | [_ : _ ⊢ _] ⇒ rewrite C_other3_neq2
        | [_ : _ ⊢ _] ⇒ rewrite C_other3_neq1'
        | [_ : _ ⊢ _] ⇒ rewrite C_other3_neq2'
     end;
 try by [].
Ltac by_or :=
 try repeat (try by [left]; right); by [].
Ltac Brick_auto := intros; e_unfold; eq_rewrite; move ⇒ i j;
 repeat match goal with
        | [_ : _ ⊢ _] ⇒ by_or
        | [j : nat ⊢ _] ⇒ induction j
        | [i : nat ⊢ _] ⇒ induction i
        | [_ : _ ⊢ _] ⇒ progress eq_rewrite
        | [_ : _ ⊢ _] ⇒ progress C_other
     end.
```

P_{22} は 3 色以上で Valid という補題.

```
Lemma Boundary_i22 : ∀ b : boundary, Boundary_i 2 2 b (e'_nm 2 2 b).
Proof.
move ⇒ b.
case.
by_or.
case.
left.
by [e_unfold].
case.
left.
by [e_unfold].
by_or.
Qed.
```

```
Lemma Boundary_j22 : ∀ b : boundary, Boundary_j 2 2 b (e_nm 2 2 b).
Proof.
move ⇒ b.
case.
by_or.
case.
left.
by [e_unfold].
case.
```

left.
by [e_unfold].
by_or.
Qed.

Lemma $Brick22\_eexx$: $\forall$ b : $boundary$,
b 0 1 == b 3 1 $\rightarrow$ b 0 2 == b 3 2 $\rightarrow$
$Brick$ 2 2 (e_nm 2 2 b) (e'_nm 2 2 b).
Proof.
$Brick\_auto$.
Qed.

Lemma $Brick22\_enee$: $\forall$ b : $boundary$,
b 0 1 == b 3 1 $\rightarrow$ b 0 2 != b 3 2 $\rightarrow$ b 1 0 == b 1 3 $\rightarrow$ b 2 0 == b 2 3 $\rightarrow$
$Brick$ 2 2 (e_nm 2 2 b) (e'_nm 2 2 b).
Proof.
$Brick\_auto$.
Qed.

Lemma $Brick22\_enen$: $\forall$ b : $boundary$,
b 0 1 == b 3 1 $\rightarrow$ b 0 2 != b 3 2 $\rightarrow$ b 1 0 == b 1 3 $\rightarrow$ b 2 0 != b 2 3 $\rightarrow$
$Brick$ 2 2 (e_nm 2 2 b) (e'_nm 2 2 b).
Proof.
$Brick\_auto$.
Qed.

Lemma $Brick22\_enne$: $\forall$ b : $boundary$,
b 0 1 == b 3 1 $\rightarrow$ b 0 2 != b 3 2 $\rightarrow$ b 1 0 != b 1 3 $\rightarrow$ b 2 0 == b 2 3 $\rightarrow$
$Brick$ 2 2 (e_nm 2 2 b) (e'_nm 2 2 b).
Proof.
$Brick\_auto$.
Qed.

Lemma $Brick22\_ennn$: $\forall$ b : $boundary$,
b 0 1 == b 3 1 $\rightarrow$ b 0 2 != b 3 2 $\rightarrow$ b 1 0 != b 1 3 $\rightarrow$ b 2 0 != b 2 3 $\rightarrow$
$Brick$ 2 2 (e_nm 2 2 b) (e'_nm 2 2 b).
Proof.
$Brick\_auto$.
Qed.

Lemma $Brick22\_neee$: $\forall$ b : $boundary$,
b 0 1 != b 3 1 $\rightarrow$ b 0 2 == b 3 2 $\rightarrow$ b 1 0 == b 1 3 $\rightarrow$ b 2 0 == b 2 3 $\rightarrow$
$Brick$ 2 2 (e_nm 2 2 b) (e'_nm 2 2 b).
Proof.
$Brick\_auto$.
Qed.

Lemma $Brick22\_neen$: $\forall$ b : $boundary$,

$b\ 0\ 1 \mathrel{!=} b\ 3\ 1 \to b\ 0\ 2 == b\ 3\ 2 \to b\ 1\ 0 == b\ 1\ 3 \to b\ 2\ 0 \mathrel{!=} b\ 2\ 3 \to$
$Brick\ 2\ 2\ (e\_nm\ 2\ 2\ b)\ (e'\_nm\ 2\ 2\ b)$.
`Proof.`
Brick_auto.
`Qed.`

`Lemma` *Brick22_nene* : $\forall\ b$: *boundary*,
$b\ 0\ 1 \mathrel{!=} b\ 3\ 1 \to b\ 0\ 2 == b\ 3\ 2 \to b\ 1\ 0 \mathrel{!=} b\ 1\ 3 \to b\ 2\ 0 == b\ 2\ 3 \to$
$Brick\ 2\ 2\ (e\_nm\ 2\ 2\ b)\ (e'\_nm\ 2\ 2\ b)$.
`Proof.`
Brick_auto.
`Qed.`

`Lemma` *Brick22_nenn* : $\forall\ b$: *boundary*,
$b\ 0\ 1 \mathrel{!=} b\ 3\ 1 \to b\ 0\ 2 == b\ 3\ 2 \to b\ 1\ 0 \mathrel{!=} b\ 1\ 3 \to b\ 2\ 0 \mathrel{!=} b\ 2\ 3 \to$
$Brick\ 2\ 2\ (e\_nm\ 2\ 2\ b)\ (e'\_nm\ 2\ 2\ b)$.
`Proof.`
Brick_auto.
`Qed.`

`Lemma` *Brick22_nnee* : $\forall\ b$: *boundary*,
$b\ 0\ 1 \mathrel{!=} b\ 3\ 1 \to b\ 0\ 2 \mathrel{!=} b\ 3\ 2 \to b\ 1\ 0 == b\ 1\ 3 \to b\ 2\ 0 == b\ 2\ 3 \to$
$Brick\ 2\ 2\ (e\_nm\ 2\ 2\ b)\ (e'\_nm\ 2\ 2\ b)$.
`Proof.`
Brick_auto.
`Qed.`

`Lemma` *Brick22_nnen* : $\forall\ b$: *boundary*,
$b\ 0\ 1 \mathrel{!=} b\ 3\ 1 \to b\ 0\ 2 \mathrel{!=} b\ 3\ 2 \to b\ 1\ 0 == b\ 1\ 3 \to b\ 2\ 0 \mathrel{!=} b\ 2\ 3 \to$
$Brick\ 2\ 2\ (e\_nm\ 2\ 2\ b)\ (e'\_nm\ 2\ 2\ b)$.
`Proof.`
Brick_auto.
`Qed.`

`Lemma` *Brick22_nnne* : $\forall\ b$: *boundary*,
$b\ 0\ 1 \mathrel{!=} b\ 3\ 1 \to b\ 0\ 2 \mathrel{!=} b\ 3\ 2 \to b\ 1\ 0 \mathrel{!=} b\ 1\ 3 \to b\ 2\ 0 == b\ 2\ 3 \to$
$Brick\ 2\ 2\ (e\_nm\ 2\ 2\ b)\ (e'\_nm\ 2\ 2\ b)$.
`Proof.`
Brick_auto.
`Qed.`

`Lemma` *Brick22_nnnn* : $\forall\ b$: *boundary*,
$b\ 0\ 1 \mathrel{!=} b\ 3\ 1 \to b\ 0\ 2 \mathrel{!=} b\ 3\ 2 \to b\ 1\ 0 \mathrel{!=} b\ 1\ 3 \to b\ 2\ 0 \mathrel{!=} b\ 2\ 3 \to$
$Brick\ 2\ 2\ (e\_nm\ 2\ 2\ b)\ (e'\_nm\ 2\ 2\ b)$.
`Proof.`
Brick_auto.
`Qed.`

```
Lemma Brick22: ∀ b : boundary, Brick 2 2 (e_nm 2 2 b) (e'_nm 2 2 b).
Proof.
move ⇒ b.
case (eq_or_neq (b 0 1) (b 3 1)) ⇒ H;
case (eq_or_neq (b 0 2) (b 3 2)) ⇒ H0;
case (eq_or_neq (b 1 0) (b 1 3)) ⇒ H1;
case (eq_or_neq (b 2 0) (b 2 3)) ⇒ H2.
apply (Brick22_eexx b H H0).
apply (Brick22_eexx b H H0).
apply (Brick22_eexx b H H0).
apply (Brick22_eexx b H H0).
apply (Brick22_enee b H H0 H1 H2).
apply (Brick22_enen b H H0 H1 H2).
apply (Brick22_enne b H H0 H1 H2).
apply (Brick22_ennn b H H0 H1 H2).
apply (Brick22_neee b H H0 H1 H2).
apply (Brick22_neen b H H0 H1 H2).
apply (Brick22_nene b H H0 H1 H2).
apply (Brick22_nenn b H H0 H1 H2).
apply (Brick22_nnee b H H0 H1 H2).
apply (Brick22_nnen b H H0 H1 H2).
apply (Brick22_nnne b H H0 H1 H2).
apply (Brick22_nnnn b H H0 H1 H2).
Qed.
```

```
Lemma P22_Valid_nm : ∀ b : boundary, Valid_nm 2 2 b.
Proof.
move ⇒ b.
repeat split.
apply Boundary_i22.
apply Boundary_j22.
apply Brick22.
Qed.
```

$m \geq 2$ で P_{2m} が Valid なら, $P_{2(m+1)}$ も Valid という補題.

```
Lemma replace_e_n2 : ∀ m : nat,
 2 ≤ m → (fun b : boundary ⇒ e_n2 (S m) b) = (fun b : boundary ⇒ (fun (i j : nat) ⇒
match i with 0 ⇒ (bSnm_to_b1m 2 b) 0 j | S i' ⇒ e_n2 m (bSnm_to_bnm 2 b) i' j end)).
Proof.
move ⇒ m H.
rewrite /e_n2.
induction m.
discriminate H.
induction m.
```

```
discriminate H.
by [].
Qed.
Lemma replace_e'_n2 : ∀ m : nat,
 2 ≤ m → (fun b : boundary ⇒ e'_n2 (S m) b) = (fun b : boundary ⇒ (fun (i j : nat) ⇒
match i with 0 ⇒ 0 | 1 ⇒ e'_1m 2 (bSnm_to_b1m 2 b) 1 j | S i' ⇒ e'_n2 m (bSnm_to_bnm
2 b) i' j end)).
Proof.
move ⇒ m H.
rewrite /e'_n2.
induction m.
discriminate H.
induction m.
discriminate H.
by [].
Qed.
Lemma Boundary_i_ind_2m :
 ∀ (b : boundary)(m : nat), 2 ≤ m →
 (∀ b' : boundary, Boundary_i 2 m b' (e'_nm 2 m b')) →
 Boundary_i 2 (S m) b (e'_nm 2 (S m) b).
Proof.
move ⇒ b m H H0.
rewrite /e'_nm/e_nm.
rewrite (replace_e_n2 m H).
induction m.
discriminate H.
induction m.
discriminate H.
clear IHm IHm0.
move : (H0 (bnm_to_bmn (bSnm_to_bnm 2 (bnm_to_bmn b)))).
rewrite /Boundary_i/e'_nm/e_nm/enm_to_emn/bnm_to_bmn/bSnm_to_b1m/bSnm_to_bnm.
move ⇒ H1 i.
induction i.
by_or.
induction i.
left.
split.
by [].
case (H1 1) ⇒ H2.
apply H2.
case H2; discriminate.
induction i.
```

```
left.
split.
by [].
case (H1 2) => H2.
apply H2.
case H2; discriminate.
by_or.
Qed.

Lemma Boundary_j_ind_2m :
 ∀ (b : boundary)(m : nat), 2 ≤ m →
 (∀ b' : boundary, Boundary_j 2 m b' (e_nm 2 m b')) →
 Boundary_j 2 (S m) b (e_nm 2 (S m) b).
Proof.
move => b m H H0.
rewrite /e'_nm/e_nm.
rewrite (replace_e'_n2 m H).
induction m.
discriminate H.
induction m.
discriminate H.
clear IHm IHm0.
move : (H0 (bnm_to_bmn (bSnm_to_bnm 2 (bnm_to_bmn b)))).
rewrite /Boundary_j/e'_nm/e_nm/enm_to_emn/bnm_to_bmn/bSnm_to_b1m/bSnm_to_bnm.
move => H1 j.
induction j.
by_or.
induction j.
rewrite /e'_1m.
by [left].
case (H1 (S j)) => H2.
left.
apply H2.
case H2 => H3.
discriminate H3.
repeat right.
apply H3.
Qed.

Lemma Brick_ind_2m :
 ∀ (b : boundary)(m : nat), 2 ≤ m →
 (∀ b' : boundary, Brick 2 m (e_nm 2 m b') (e'_nm 2 m b')) →
 Brick 2 (S m) (e_nm 2 (S m) b) (e'_nm 2 (S m) b).
Proof.
```

```
move ⇒ b m H H0.
rewrite /e'_nm/e_nm.
rewrite (replace_e_n2 m H).
rewrite (replace_e'_n2 m H).
induction m.
discriminate H.
induction m.
discriminate H.
clear IHm IHm0.
move : (H0 (bnm_to_bmn (bSnm_to_bnm 2 (bnm_to_bmn b)))).
rewrite /Brick/e'_nm/e_nm/enm_to_emn/bnm_to_bmn/bSnm_to_b1m/bSnm_to_bnm.
move ⇒ H1 i j.
induction m.
induction j.
induction i.
rewrite /e_n2/e_22/e'_1m.
C_other.
by_or.
induction i.
rewrite /e_n2/e_22/e'_1m.
C_other.
by_or.
by_or.
apply H1.
induction j.
induction i.
rewrite /e_n2/e_22/e'_1m/bSnm_to_b1m.
C_other.
by_or.
induction i.
rewrite /e_n2/e_22/e'_1m/bSnm_to_b1m.
C_other.
by_or.
by_or.
apply H1.
Qed.
Lemma Valid_nm_ind_2m : ∀ (b : boundary)(m : nat),
 2 ≤ m → (∀ b' : boundary, Valid_nm 2 m b') → Valid_nm 2 (S m) b.
Proof.
move ⇒ b m H H0.
split.
apply (Boundary_i_ind_2m _ _ H).
```

```
apply H0.
split.
apply (Boundary_j_ind_2m _ _ H).
apply H0.
apply (Brick_ind_2m _ _ H).
apply H0.
Qed.
```

$m \geq 2$ なら, P_{2m} は 3 色以上で Valid という補題.

```
Lemma P2m_Valid_nm : ∀ (b : boundary)(m : nat), 2 ≤ m → Valid_nm 2 m b.
Proof.
induction m.
discriminate.
induction m.
discriminate.
clear IHm IHm0.
move : b.
induction m.
move ⇒ b H.
apply P22_Valid_nm.
move ⇒ b H.
apply Valid_nm_ind_2m.
apply H.
move ⇒ b'.
apply IHm.
apply H.
Qed.
```

$n, m \geq 2$ で P_{nm} が Tileable なら, $P_{(n+1)m}$ も Tileable という補題.

```
Lemma replace_e_nm : ∀ (b : boundary)(n m : nat),
 e_nm n.+3 m b = (fun (i j : nat) ⇒ match i with 0 ⇒ (bSnm_to_b1m m b) 0 j | S i' ⇒
e_nm n.+2 m (bSnm_to_bnm m b) i' j end).
Proof.
move ⇒ b n m.
by [rewrite /e_nm].
Qed.
```

```
Lemma replace_e'_nm : ∀ (b : boundary)(n m : nat),
 e'_nm n.+3 m b = (fun (i j : nat) ⇒ match i with 0 ⇒ 0 | 1 ⇒ e'_1m m (bSnm_to_b1m
m b) 1 j | S i' ⇒ e'_nm n.+2 m (bSnm_to_bnm m b) i' j end).
Proof.
move ⇒ b n m.
by [rewrite /e'_nm].
Qed.
```

```
Lemma Boundary_i_ind_nm :
 ∀ (b : boundary)(n m : nat), 2 ≤ n → 2 ≤ m →
 (∀ b' : boundary, Boundary_i n m b' (e'_nm n m b')) →
 Boundary_i (S n) m b (e'_nm (S n) m b).
Proof.
move ⇒ b n m H H0 H1.
induction n.
discriminate H.
induction n.
discriminate H.
clear IHn IHn0.
move : (H1 (bSnm_to_bnm m b)).
rewrite /Boundary_i.
move ⇒ H2.
induction i.
by_or.
move : (H2 i).
rewrite replace_e'_nm.
induction m.
discriminate H0.
induction m.
discriminate H0.
clear IHm IHm0.
case ⇒ H3.
left.
induction i.
by [rewrite /e'_1m/bSnm_to_b1m/bSnm_to_bnm].
move : H3.
by [rewrite /bSnm_to_bnm].
case H3 ⇒ H4.
rewrite H4.
rewrite /e'_1m/bSnm_to_b1m/bSnm_to_bnm.
by_or.
repeat right.
apply H4.
Qed.
```

```
Lemma Boundary_j_ind_nm :
 ∀ (b : boundary)(n m : nat), 2 ≤ n → 2 ≤ m →
 (∀ b' : boundary, Boundary_j n m b' (e_nm n m b')) →
 Boundary_j (S n) m b (e_nm (S n) m b).
Proof.
move ⇒ b n m H H0 H1.
```

```
induction n.
discriminate H.
induction n.
discriminate H.
clear IHn IHn0.
move : (H1 (bSnm_to_bnm m b)).
rewrite /Boundary_j.
move => H2 j.
move : (H2 j).
induction j.
by_or.
rewrite replace_e_nm.
induction m.
discriminate H0.
induction m.
discriminate H0.
clear IHm IHm0.
case => H4.
left.
split.
by [rewrite /bSnm_to_bnm/bSnm_to_b1m].
rewrite /bSnm_to_bnm/bSnm_to_b1m in H4.
apply H4.
right.
apply H4.
Qed.
Lemma Brick_ind_nm :
 ∀ (b : boundary)(n m : nat), 2 ≤ n → 2 ≤ m →
 (∀ b' : boundary, Valid_nm n m b') →
 Brick (S n) m (e_nm (S n) m b) (e'_nm (S n) m b).
Proof.
move => b n m H H0 H1.
induction n.
discriminate H.
induction n.
discriminate H.
clear IHn IHn0.
move : (H1 (bSnm_to_bnm m b)).
rewrite /Valid_nm/Boundary_i/Boundary_j/Brick.
elim => H2.
elim => H3 H4.
clear H2.
```

```
rewrite replace_e_nm replace_e'_nm.
move ⇒ i j.
induction i.
case (H3 j.+1) ⇒ H5.
elim H5 ⇒ H6 H7.
rewrite (elimTF eqP H6).
induction m.
discriminate H0.
induction m.
discriminate H0.
clear IHm IHm0.
rewrite /bSnm_to_b1m/bSnm_to_bnm/enm_to_emn/bnm_to_bmn/e'_1m.
induction j.
C_other.
by_or.
induction j.
C_other.
by_or.
C_other.
by_or.
case H5 ⇒ H6.
discriminate H6.
repeat right.
apply H6.
apply H4.
Qed.
```

```
Lemma Valid_nm_ind_nm : ∀ (b : boundary)(n m : nat),
 2 ≤ n → 2 ≤ m → (∀ b' : boundary, Valid_nm n m b') → Valid_nm (S n) m b.
Proof.
move ⇒ b n m H H0 H1.
split.
apply (Boundary_i_ind_nm _ _ _ H H0).
apply H1.
split.
apply (Boundary_j_ind_nm _ _ _ H H0).
apply H1.
apply (Brick_ind_nm _ _ _ H H0).
apply H1.
Qed.
```

$n, m \geq 2$ なら, P_{nm} は 3 色以上で Tileable という補題.

```
Theorem e_nm_Valid : ∀ (b : boundary)(n m : nat),
 2 ≤ n → 2 ≤ m → Valid n m b (e_nm n m b) (e'_nm n m b).
```

```
Proof.
move => b n m H H0.
induction n.
discriminate H.
induction n.
discriminate H.
clear IHn IHn0.
induction m.
discriminate H0.
induction m.
discriminate H0.
clear IHm IHm0.
move : b.
induction n.
move => b.
apply (P2m_Valid_nm _ _ H0).
move => b.
apply Valid_nm_ind_nm.
apply H.
apply H0.
apply IHn.
apply H.
Qed.
```

```
Theorem P22_Tileable : ∀ (b : boundary)(n m : nat),
  2 ≤ n → 2 ≤ m → ∃ (e e' : edge), Valid n m b e e'.
Proof.
move => b n m H0 H1.
∃ (e_nm n m b).
∃ (e'_nm n m b).
apply (e_nm_Valid b n m H0 H1).
Qed.
```

1.5 Export to Mathematica

Mathematica へのエクスポートのための設定

```
Definition null_list {A : Type} (l m : list A) : Prop.
Proof.
apply True.
Qed.
Notation "{ x }" := (cons x nil).
Notation "{ x , .. , y }" := (cons x .. (cons y nil) ..).
```

```
Notation "Tiling[ l , m ]" := (null_list l m).
Fixpoint e_list_n (f : nat → nat)(n : nat) :=
 match n with
   | 0 ⇒ nil
   | S i ⇒ (e_list_n f i) ++ {f (S i)}
 end.
Fixpoint e_list (e : edge)(n m : nat) :=
 match n with
   | 0 ⇒ {e_list_n (e 0) m}
   | S i ⇒ (e_list e i m) ++ {e_list_n (e (S i)) m}
 end.
Fixpoint e'_list_n (f : nat → nat)(n : nat) :=
 match n with
   | 0 ⇒ {f 0}
   | S i ⇒ (e'_list_n f i) ++ {f (S i)}
 end.
Fixpoint e'_list (e : edge)(n m : nat) :=
 match n with
   | 0 ⇒ nil
   | S i ⇒ (e'_list e i m) ++ {e'_list_n (e (S i)) m}
 end.
Definition tiling_nm2 (n m : nat)(b : boundary) :=
 Tiling[e_list (e_nm n m b) n m, e'_list (e'_nm n m b) n m].
```

```
Compute (tiling_nm2 4 4 (fun i j ⇒ match j with 0 ⇒ match i with 2 | 3 ⇒ 3 | _ ⇒ 4 end
| 1 ⇒ 2 | 3 ⇒ 1 | _ ⇒ match i with 0 ⇒ 0 | _ ⇒ 5 end end)).
```

どうしても = と : Prop が邪魔という人向け

```
Ltac print := compute; match goal with ⊢ ?x ⇒ idtac x end.
Goal (tiling_nm2 4 4 (fun i j ⇒ match j with 0 ⇒ match i with 2 | 3 ⇒ 3 | _ ⇒ 2 end | 1
⇒ 2 | 3 ⇒ 1 | _ ⇒ match i with 0 ⇒ 0 | _ ⇒ 1 end end)).
print.
Abort.
```

```
Extraction "TilingProgram.ml" tiling_nm boundary22a boundary22b boundary22c bound-
ary44a boundary44b boundary44c.
```