Proofs are Programs

Structured Data

Short Recap

Proofs by

Case Analysis

```
destruct n as [| n'] eqn:E.
```

Induction

```
induction n as [| [n’ H]].
```

Introduction Patterns

```
destruct c as [ | | p ].

intros [ | | [] ].
```

“Informal Proofs”

Proof: by induction on n.

Proofs within Proofs

```
assert (H:..).
  { … }
```

Pairs of Numbers

Lists (of Natural Numbers)

```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).
```

```
cons 1 (cons 2 (cons 3 nil))

1 :: (2 :: (3 :: nil))

1 :: 2 :: 3 :: nil

[1;2;3]
```

Lists (of Natural Numbers)

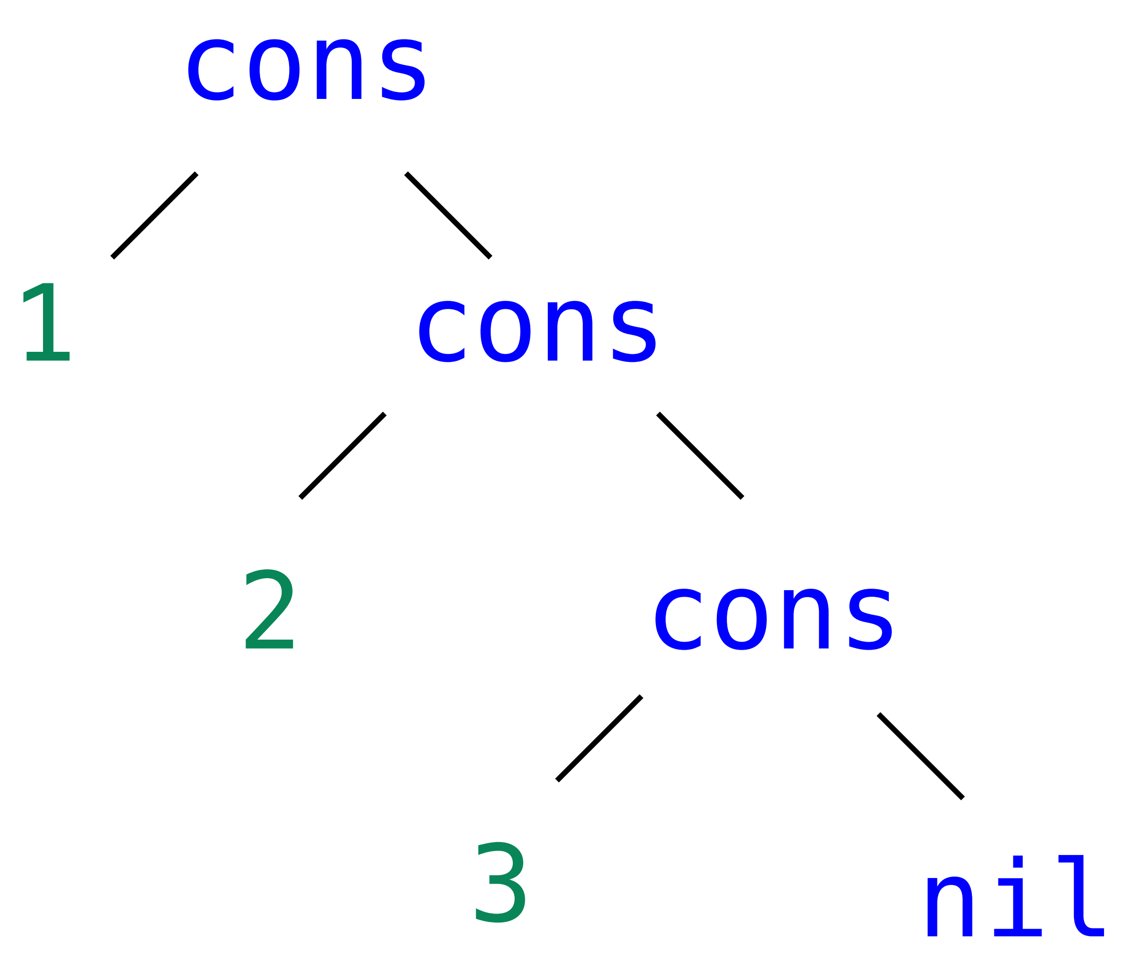
```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).
```

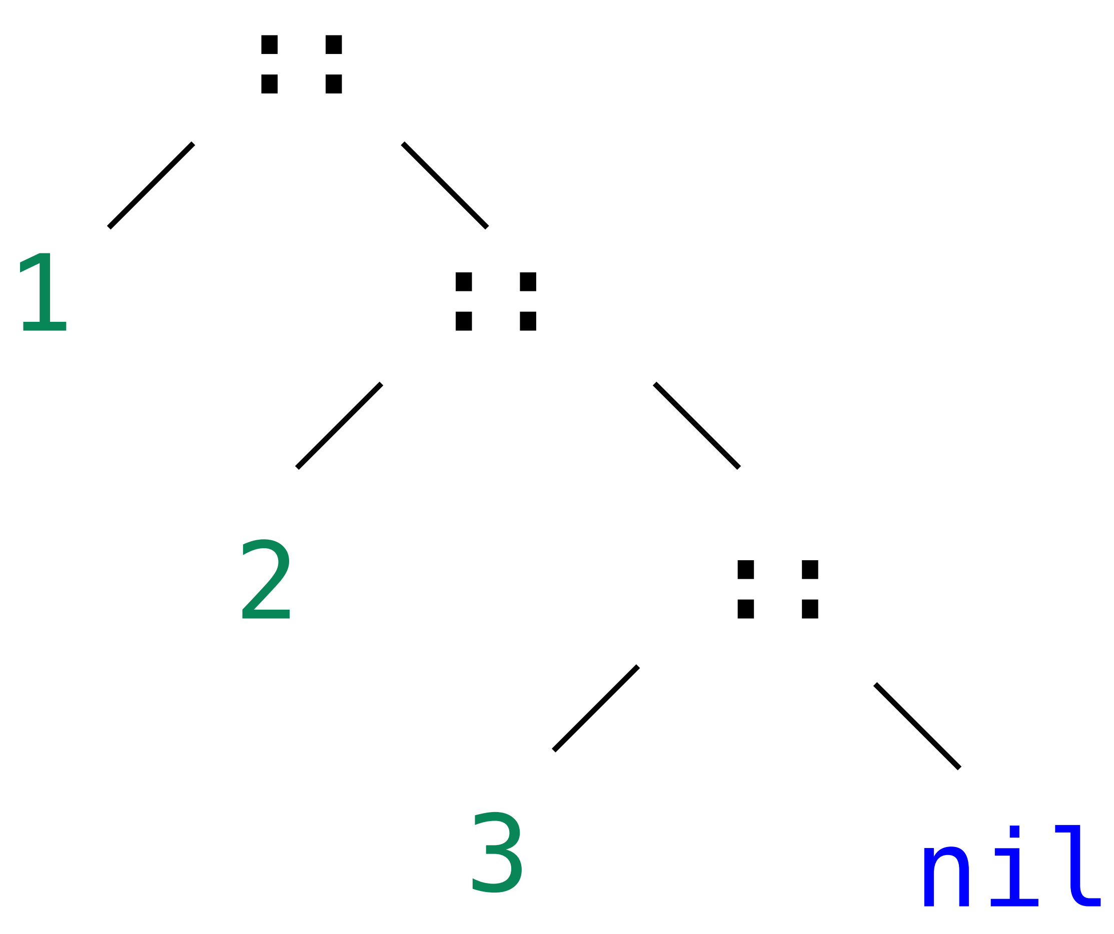
```
cons 1 (cons 2 (cons 3 nil))

1 :: (2 :: (3 :: nil))

1 :: 2 :: 3 :: nil

[1;2;3]
```





Building Lists Recursively

```
repeat 42 2
```

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

Building Lists Recursively

```
repeat 42 2
```

```
repeat 42 (S (S O))
```

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

Building Lists Recursively

```
repeat 42 2
```

```
repeat 42 (S (S O))
```

```
    ::
   /  \
 42    repeat 42 (S O)
```

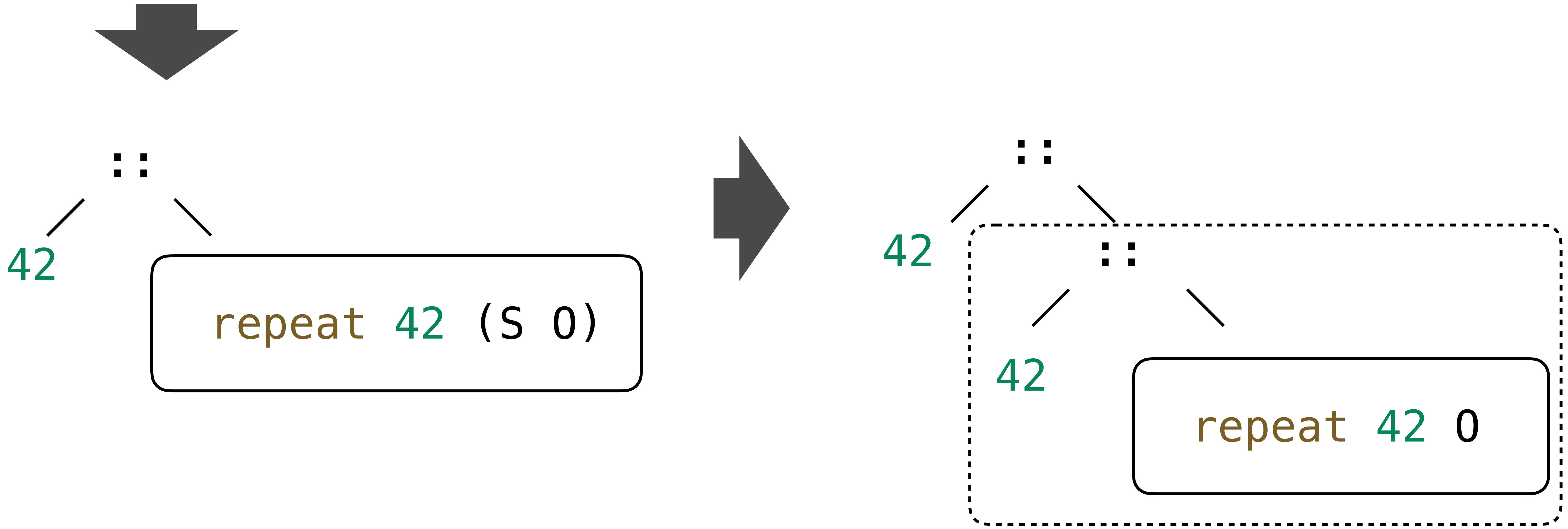
```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

Building Lists Recursively

```
repeat 42 2
```

```
repeat 42 (S (S O))
```

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

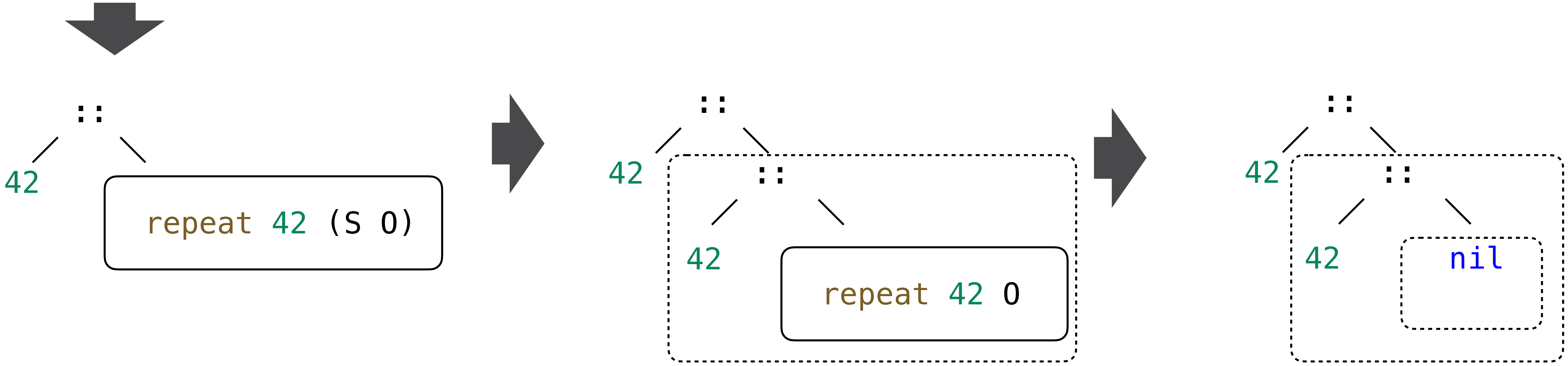


Building Lists Recursively

```
repeat 42 2
```

```
repeat 42 (S (S O))
```

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

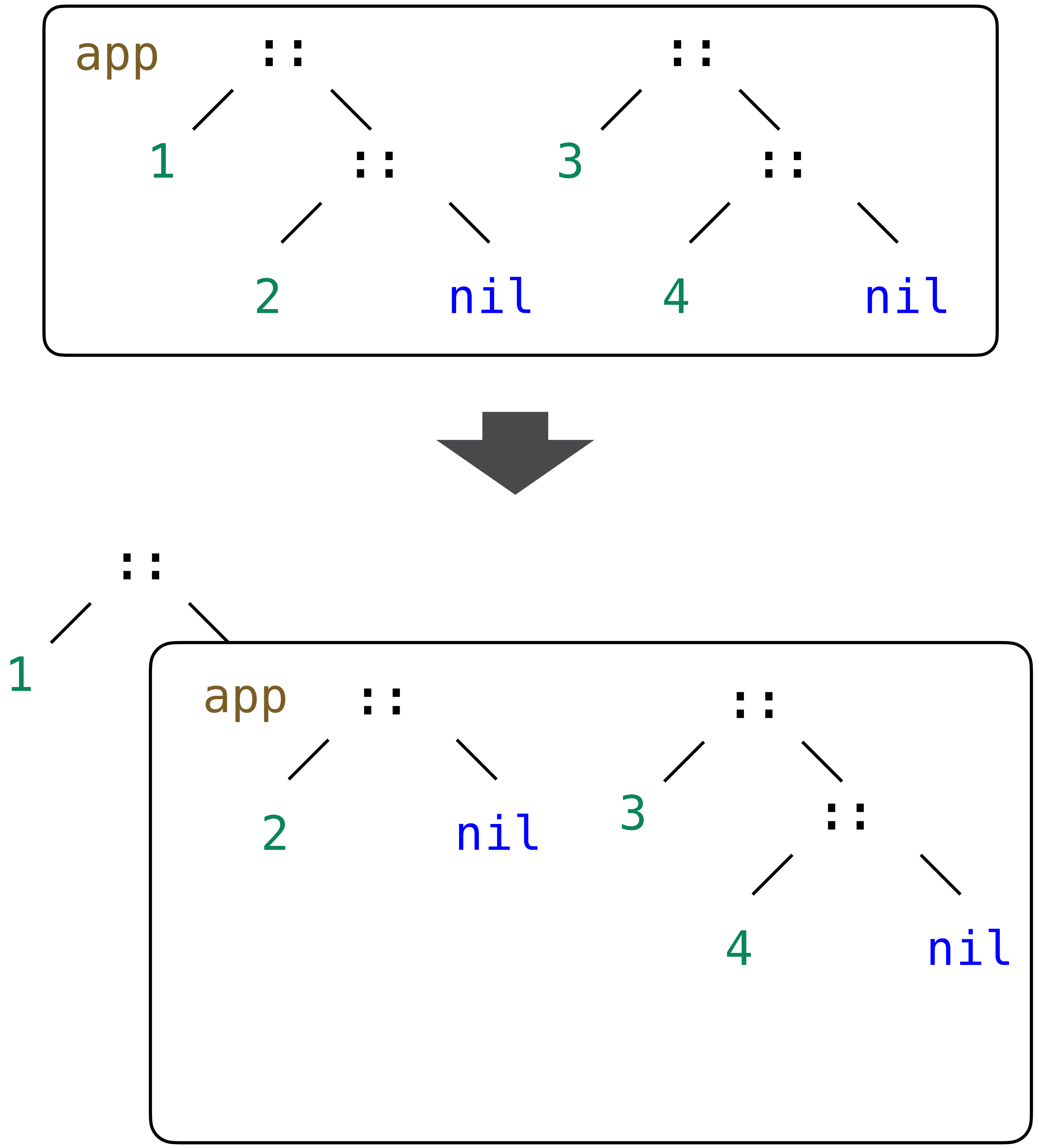


Recursive Functions on Lists

```
app    ::                ::
      /  \              /  \
     1    ::           3    ::
         /  \              /  \
        2    nil          4    nil
```

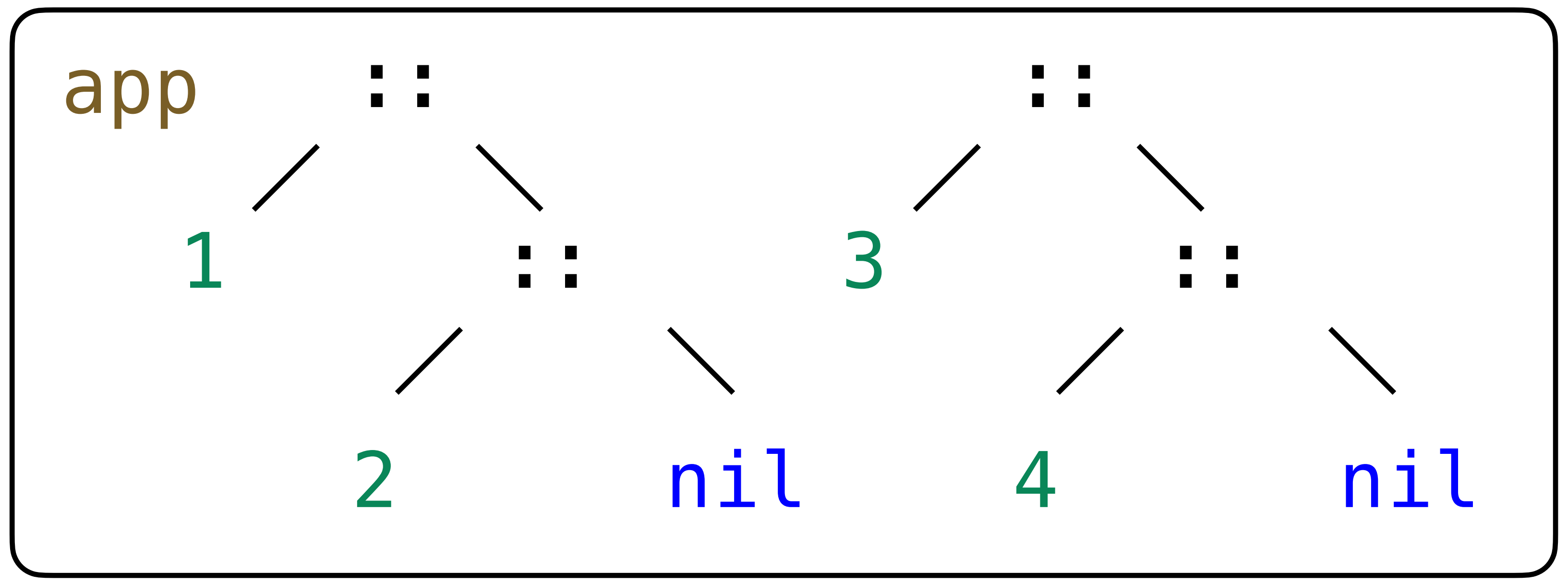
```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil    => l2
  | h :: t => h :: (app t l2)
  end.
```

Recursive Functions on Lists

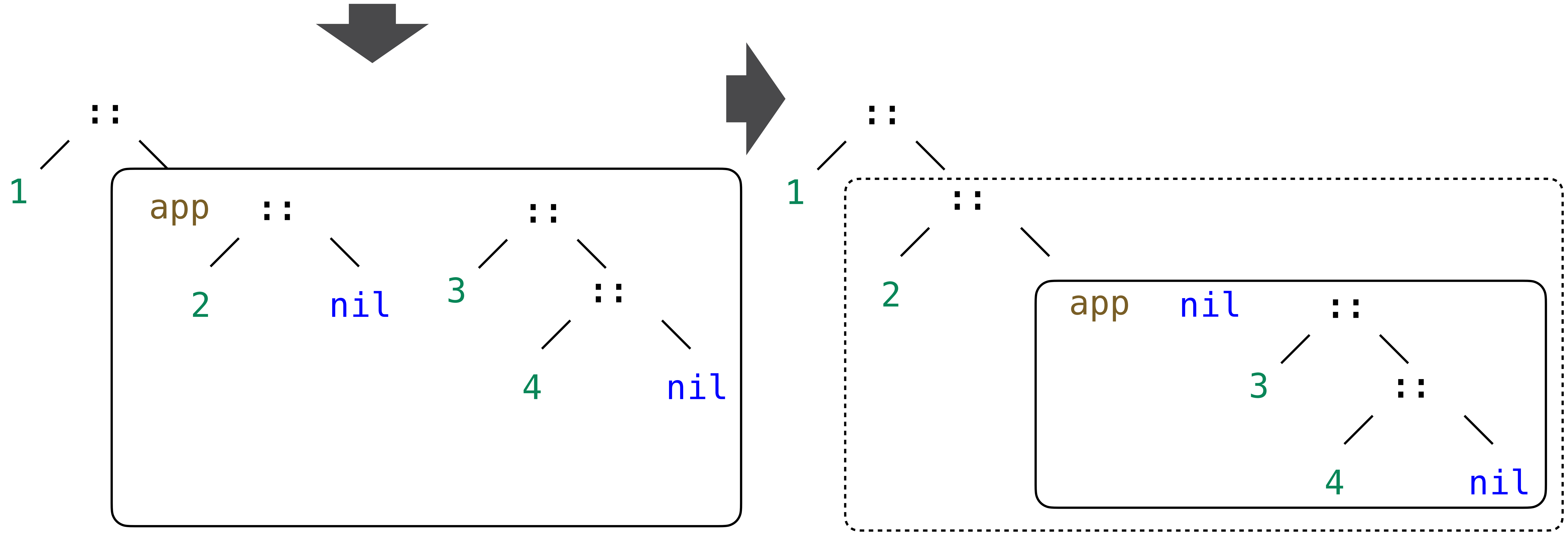



```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil    => l2
  | h :: t => h :: (app t l2)
  end.
```

Recursive Functions on Lists




```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil    => l2
  | h :: t => h :: (app t l2)
  end.
```



Recursive Functions on Lists

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil    => l2
  | h :: t => h :: (app t l2)
  end.
```

http://etc.ch/Tkpd

Quiz

```
Fixpoint foo (n : nat) : natlist :=
  match n with
  | 0 => nil
  | S n' => n :: (foo n')
  end.
```

Induction on Lists

- General principle to show that $\forall \ell \,.\, P(\ell)$
  - show $P(\text{nil})$
  - show that for any $\ell = x :: \ell'$ if $P(\ell')$ holds, then so does $P(\ell)$
- Example $P(\ell) := \ell ++ \text{nil} = \ell$
- Example $P(\ell_1) := \forall \ell_2\ \ell_3 \,.\, (\ell_1 ++ \ell_2) ++ \ell_3 = \ell_1 ++ (\ell_2 ++ \ell_3)$

Example: Induction on Lists

$$\forall \ell_1\ \ell_2\ \ell_3\,.\ (\ell_1 +\!+\ \ell_2) +\!+\ \ell_3 = \ell_1 +\!+\ (\ell_2 +\!+\ \ell_3)$$

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil    => l2
  | h :: t => h :: (app t l2)
  end.
```

Example: Induction on Lists

- To show: $\forall \ell_1\ \ell_2\ \ell_3\,.\ (\ell_1 +\!+\ \ell_2) +\!+\ \ell_3 = \ell_1 +\!+\ (\ell_2 +\!+\ \ell_3)$
- Proof: By induction on ℓ_1
  - Suppose that $\ell_1 = []$.
    We show: $([] +\!+\ \ell_2) +\!+\ \ell_3 = [] +\!+\ (\ell_2 +\!+\ \ell_3)$
    This holds as: $([] +\!+\ \ell_2) +\!+\ \ell_3 = \ell_2 +\!+\ \ell_3 = [] +\!+\ (\ell_2 +\!+\ \ell_3)$ by definition of ++
  - Suppose that $\ell_1 = n :: \ell_1'$ with $(\ell_1' +\!+\ \ell_2) +\!+\ \ell_3 = \ell_1' +\!+\ (\ell_2 +\!+\ \ell_3)$ (IH).
    We show: $((n :: \ell_1') +\!+\ \ell_2) +\!+\ \ell_3 = (n :: \ell_1') +\!+\ (\ell_2 +\!+\ \ell_3)$
    This holds as
    $((n :: \ell_1') +\!+\ \ell_2) +\!+\ \ell_3$
    $= (n :: (\ell_1' +\!+\ \ell_2)) +\!+\ \ell_3$ by definition of ++
    $= n :: ((\ell_1' +\!+\ \ell_2) +\!+\ \ell_3)$ by definition of ++
    $= n :: (\ell_1' +\!+\ (\ell_2 +\!+\ \ell_3))$ by IH
    $= (n :: \ell_1') +\!+\ (\ell_2 +\!+\ \ell_3)$ by definition of ++

Generalization/Strengthening

$$\forall c\ n\,.\ \text{repeat}\ n\ c \mathrel{++} \text{repeat}\ c\ n = \text{repeat}\ n\ (c + c)$$

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil    => l2
  | h :: t => h :: (app t l2)
  end.
```

Generalization/Strengthening

- To show: $\forall\ n\ c_1\ c_2$. repeat $n\ c_1\ \ ++$ repeat $n\ c_2 =$ repeat $n\ (c_1 + c_2)$
- Proof: By induction on c_1
  - Suppose that $c_1 = 0$.
    We show: repeat $n\ 0 ++$ repeat $n\ c_2\ =$ repeat $n\ (0 + c_2)$
    This holds as: repeat $n\ 0 ++$ repeat $n\ c_2 = [] ++$ repeat $n\ c_2 =$ repeat $n\ c_2 =$ repeat $n\ (0 + c_2)$
    by definitions of ++ and +
  - Suppose that $c_1 = S\ c_1'$ with repeat $n\ c_1' ++$ repeat $n\ c_2\ =$ repeat $n\ (c_1' + c_2)$ (IH).
    We show: repeat $n\ (S\ c_1')\ \ ++$ repeat $n\ c_2 =$ repeat $n\ ((S\ c_1') + c_2)$
    This holds as
    repeat $n\ (S\ c_1') ++$ repeat $n\ c_2$
    $= (n ::$ repeat $n\ c_1') ++$ repeat $n\ c_2$ by definition of repeat
    $= n ::$ (repeat $n\ c_1' ++$ repeat $n\ c_2$) by definition of ++
    $= n ::$ (repeat $n\ (c_1' + c_2)$) by IH
    $=$ repeat $n\ (S\ (c_1' + c_2))$ by definition of repeat
    $=$ repeat $n\ ((S\ c_1') + c_2)$ by definition of +

Stepping Back and Revising

$$\forall \ell \,.\, \text{length } (\text{rev } \ell) = \text{length } \ell$$

```
Fixpoint length (l:natlist) : nat :=
  match l with
  | nil => O
  | h :: t => S (length t)
  end.
```

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil    => nil
  | h :: t => rev t ++ [h]
  end.
```

Stepping Back + Revising

- To show: $\forall \ell\ .\ \text{length}\ (\text{rev}\ \ell) = \text{length}\ \ell$

- Proof: By induction on ℓ

  - Suppose that $\ell = []$.
    We show: $\text{length}\ (\text{rev}\ []) = \text{length}\ []$
    This holds as: $\text{length}\ (\text{rev}\ []) = \text{length}\ [] = \text{length}\ []$ by definition of rev

  - Suppose that $\ell = n :: \ell'$ with $\text{length}\ (\text{rev}\ \ell') = \text{length}\ \ell'$ (IH).
    We show: $\text{length}\ (\text{rev}\ (n :: \ell')) = \text{length}\ (n :: \ell')$
    This holds as
    $\text{length}\ (\text{rev}\ (n :: \ell'))$
    $= \text{length}\ (\text{rev}\ \ell' ++ [n])$ by definition of rev
    $= 1 + \text{length}\ (\text{rev}\ \ell')$ by Lemma
    $= 1 + \text{length}\ \ell'$ by IH
    $= \text{length}\ (n :: \ell')$ by definition of length

http://etc.ch/Tkpd

Quiz

```
Theorem foo1 : forall n:nat, forall l:natlist,
  repeat n 0 = l -> length l = 0.
```

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

```
Fixpoint length (l:natlist) : nat :=
  match l with
  | nil => O
  | h :: t => S (length t)
  end.
```

Quiz

```
Theorem foo2 :  forall n m : nat,
length (repeat n m) = m.
```

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

```
Fixpoint length (l:natlist) : nat :=
  match l with
  | nil => O
  | h :: t => S (length t)
  end.
```

Option Types

Partial Maps

http://etc.ch/Tkpd

Quiz

```
Theorem quiz1 : forall (d : partial_map)
                       (x : id) (v: nat),
  find x (update d x v) = Some v.
```

```
Definition update (d : partial_map)
                  (x : id) (value : nat)
                  : partial_map :=
  record x value d.
```

```
Fixpoint find (x : id) (d : partial_map) : natoption :=
  match d with
  | empty         => None
  | record y v d' => if eqb_id x y
                     then Some v
                     else find x d'
  end.
```

http://etc.ch/Tkpd

Quiz

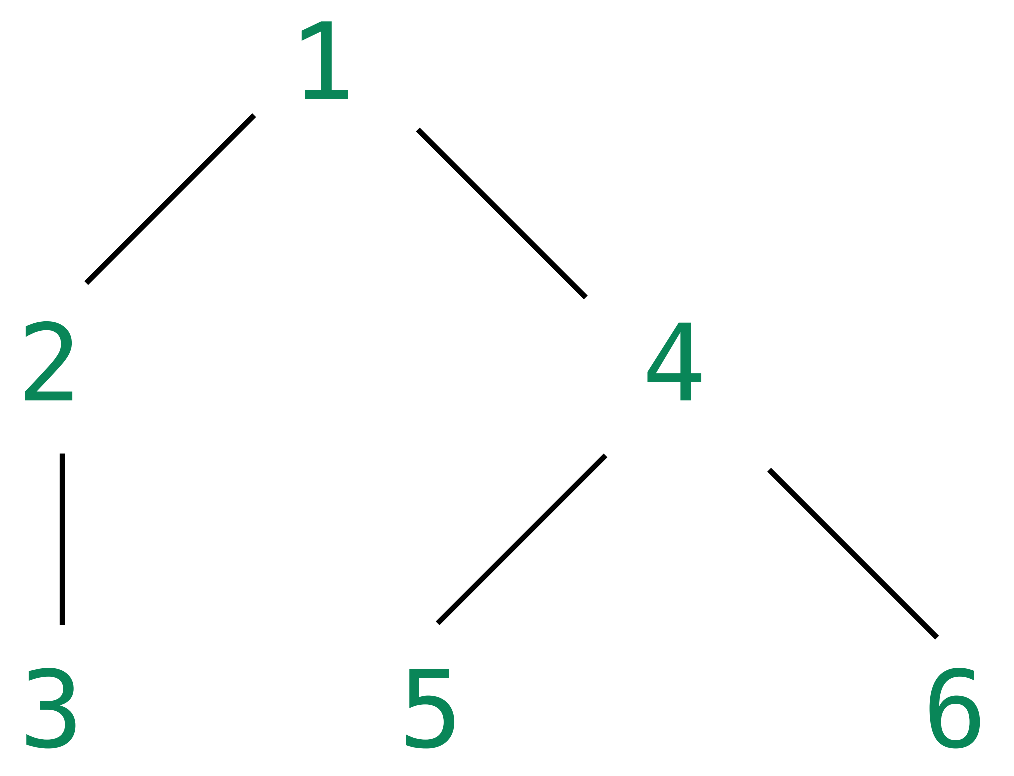
```
Theorem quiz2 : forall (d : partial_map)
                       (x y : id) (o: nat),
  eqb_id x y = false ->
  find x (update d y o) = find x d.
```

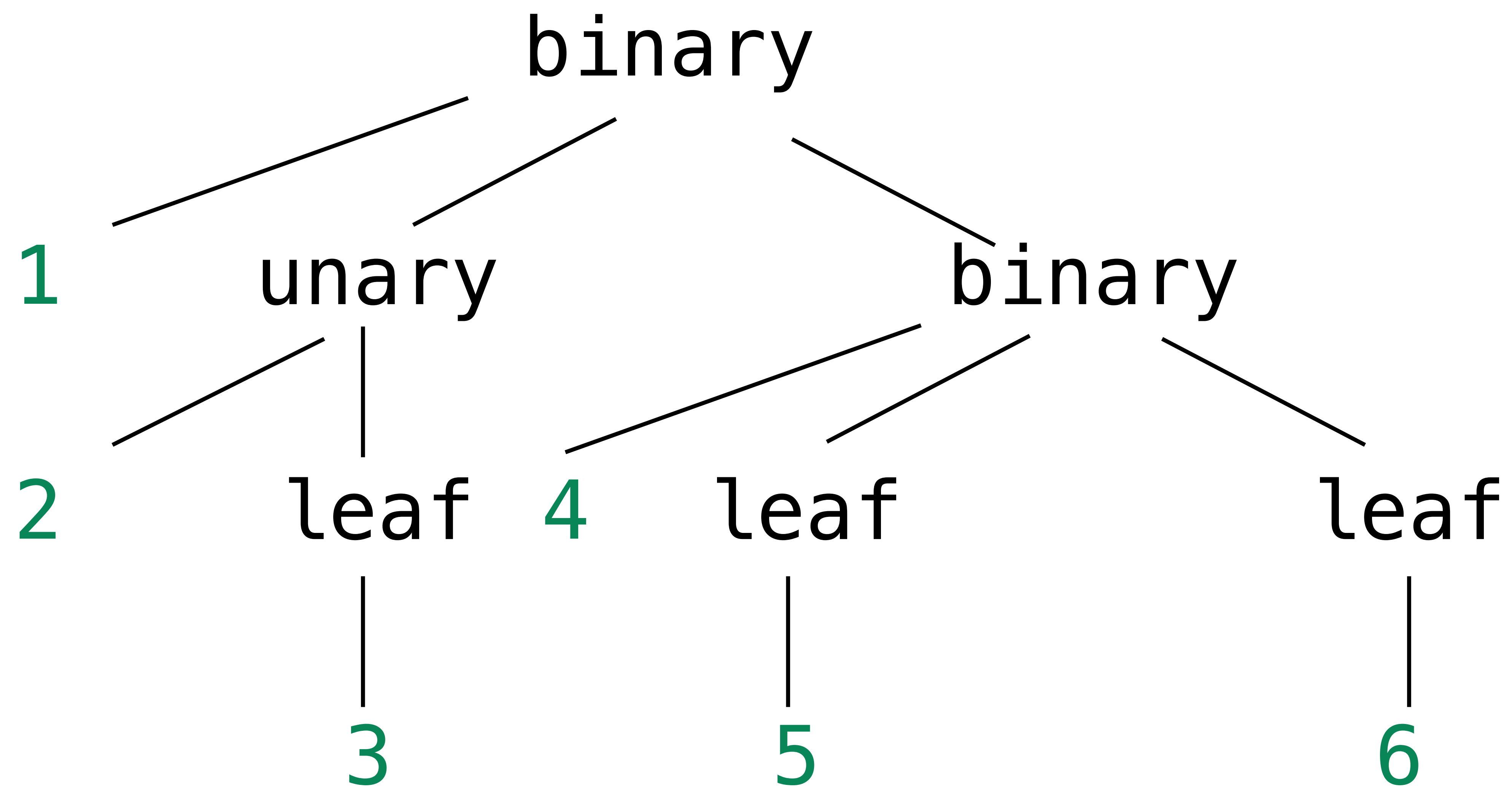
```
Definition update (d : partial_map)
                  (x : id) (value : nat)
                  : partial_map :=
  record x value d.
```

```
Fixpoint find (x : id) (d : partial_map) : natoption :=
  match d with
  | empty         => None
  | record y v d' => if eqb_id x y
                     then Some v
                     else find x d'
  end.
```

Trees of Natural Numbers

```
Inductive btree : Type :=
  | leaf (n: nat)
  | unary (n: nat) (t: btree)
  | binary (n: nat) (t1: btree) (t2: btree).
```





Induction on Trees

```
Inductive btree : Type :=
  | leaf (n: nat)
  | unary (n: nat) (t: btree)
  | binary (n: nat) (t1: btree) (t2: btree).
```

- General principle to show that $\forall t \,.\, P(t)$
  - show $P(\text{leaf } n)$
  - show that for any $t = \text{unary } n \ t'$ if $P(t')$ holds, then so does $P(t)$
  - show that for any $t = \text{binary } n \ t_1' \ t_2'$ if $P(t_1')$ and $P(t_2')$ hold, then so does $P(t)$

Tree Induction

$$\forall t \,.\; \text{sumLabels}\ (\text{incrementLabels}\ t) = (\text{size}\ t) + \text{sumLabels}\ t$$

```
Fixpoint size (t: btree): nat :=
  match t with
  | leaf n => 1
  | unary n t => 1 + size t
  | binary n t1 t2 => 1 + size t1 + size t2
  end.
```

```
Fixpoint sumLabels (t: btree): nat :=
  match t with
  | leaf n => n
  | unary n t => n + sumLabels t
  | binary n t1 t2 => n + sumLabels t1
                        + sumLabels t2
  end.
```

```
Fixpoint incrementLabels (t: btree): btree :=
  match t with
  | leaf n => leaf (S n)
  | unary n t => unary (S n) (incrementLabels t)
  | binary n t1 t2 => binary (S n) (incrementLabels t1) (incrementLabels t2)
  end.
```

Tree Induction

- To show: $\forall t\,.$ sumLabels (incrementLabels t) = (size t) + sumLabels t
- Proof: By induction on t
  - Suppose that $t =$ leaf n.
    We show: sumLabels (incrementLabels (leaf n)) = (size (leaf n)) + sumLabels (leaf n)
    This holds as:
    sumLabels (incrementLabels (leaf n)) = sumLabels (leaf $(S\ n)$) $= S\ n = 1 + n =$ (size (leaf n)) + sumLabels (leaf n)
    by definitions of incrementLabels, sumLabels and size
  - Suppose that $t =$ unary $n\ t'$ with sumLabels (incrementLabels t') = (size t') + sumLabels t' (IH).
    We show: sumLabels (incrementLabels (unary $n\ t'$)) = size (unary $n\ t'$) + sumLabels (unary $n\ t'$)
    This holds as
    sumLabels (incrementLabels (unary $n\ t'$))
    = sumLabels (unary $(S\ n)$ (incrementLabels t')) by definition of incrementLabels
    = $(S\ n)$ + sumLabels (incrementLabels t') by definition of sumLabels
    = $(S\ n)$ + (size t') + sumLabels t' by IH
    = 1 + size t' + (n + sumLabels t') by arithmetic rules
    = size (unary $n\ t'$) + sumLabels (unary $n\ t'$) by definition of size and sumLabels

Tree Induction (continued)

- To show: $\forall t\,.$ sumLabels (incrementLabels t) $=$ (size t) $+$ sumLabels t
    - ...
    - Suppose that $t =$ binary $n\ t_1'\ t_2'$ with
      sumLabels (incrementLabels t_1') $=$ (size t_1') $+$ sumLabels t_1' (IH1) and
      sumLabels (incrementLabels t_2') $=$ (size t_2') $+$ sumLabels t_2' (IH2)
      We show: sumLabels (incrementLabels (binary $n\ t_1'\ t_2'$)) $=$ size (binary $n\ t_1'\ t_2'$) $+$ sumLabels (binary $n\ t_1'\ t_2'$)
      This holds as
      sumLabels (incrementLabels (binary $n\ t_1'\ t_2'$))
      $=$ sumLabels (binary $(S\ n)$ (incrementLabels t_1') (incrementLabels t_2')) by definition of incrementLabels
      $= (S\ n) +$ (sumLabels incrementLabels t_1') $+$ sumLabels (incrementLabels t_2') by definition of sumLabels
      $= (S\ n) +$ (size t_1' $+$ sumLabels t_1') $+$ (size t_2' $+$ sumLabels t_2') by IH1 and IH2
      $= 1 +$ size t_1' $+$ size t_2' $+ (n +$ sumLabels t_1' $+$ sumLabels t_2') by arithmetics
      $=$ size (binary $n\ t_1'\ t_2'$) $+$ sumLabels (binary $n\ t_1'\ t_2'$) by definition of size and sumLabels

Summary

Datatypes on top of Natural Numbers

- Pairs

```
Inductive natprod : Type :=
  | pair (n1 n2 : nat).
```

- Lists

```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).
```

- Options

```
Inductive natoption : Type :=
  | Some (n : nat)
  | None.
```

- Partial Maps

```
Inductive partial_map : Type :=
  | empty
  | record (i : id) (v : nat) (m : partial_map).
```

- Trees

```
Inductive btree : Type :=
  | leaf (n: nat)
  | unary (n: nat) (t: btree)
  | binary (n: nat) (t1: btree) (t2: btree).
```

Induction on Datatypes

```
induction l as [| n l' IHl']
```

```
induction t as
[n | n t' IHt' | n t1' IHt1' t2' IHt2']
```

Generalising Statements

c’

$$\forall c\ n.\ \text{repeat}\ c\ n ++ \text{repeat}\ \cancel{c}\ n = \text{repeat}\ (c + \cancel{c})\ n$$

c’