Lecture 5

Logistic

Midterm: Today at 5pm CST
Due next Friday (July 10 11:59pm) CST

No need to write any check-expect.

66/56
Total: 56
Total + Extra Credit: 72

20 → 22

Map, Filter, Foldl, Foldr

Map

(map f my-list)

Input1 → function

Input2 → list

(list 1 2 3 4)

f is a function that adds 1 to any number. $f(n) = n+1$

$f(2) = f(3)$

(map f (list 1 2 3 4))

(list f(1) f(2) f(3) f(4))

(list 2 3 4 5)

```
(: add1 : Integer → Integer)
;; Adds one to any integer

(define (add1 x)
  (+ x 1))
```

```
(map add1 (list 1 2 3 4))
  ⇒ '(2 3 4 5)
```

Write a function that takes a list as input and adds to every element.

```
(: add1-list : (Listof Integer) -> (Listof Integer))

(define (add1-list my-list)
  (map add1 my-list))
```

Example: Append ! to a every string in a list

```
(: append-! : String -> String)
(define (append-! input)
  (string-append input "!"))

(map append-! (list "no" "good"))
```

⇒ (list "no!" "good!").

Write a program which takes a list of Point and outputs the list x-coordinate.

```
(: x-cor : (Listof Point) -> (Listof Real))

(define (x-cor my-list)
  (map Point-x my-list))
```

For this to work the structure Point has to be defined before.

Filter

(filter f? my-list)

↓

Boolean (Its output is a boolean)

Example

Write a function which takes list of integers and outputs a list containing only the odd integers.

```
(: my-odd? : Integer → Boolean)
(define (my-odd? num)
(if (= (remainder num 2) 1)
    #t
    #f ))
```

```
(: list-odd : (Listof Integer)
   -> (Listof Integer))
(define (list-odd my-list)
(filter my-odd? my-list))
```

Output

(filter my-odd? (list 1 2 3 4 5))

⇒

(my-odd? 1) (my-odd? 2)

#t false.

Apply (my-odd? to every element)

⇒ '(1 3 5)

(list 1 2 3 4 5)

(my-odd? 1) ⇓ #t

(my-odd? 2) ⇓ #f

(my-odd? 3) #t

(my-odd? 4) #f

(my-odd? 5) #t

(1 3 5)

Recursive idea is this

Take the first element
→ check if (func (first my-list))
is true

→ Recursively call on
the (rest my-list)

(1 2 3 4 5)

1 (2 3 4 5)

↓ Taken care by the recursion

Since 1 is (3 5)
odd we will
add to our list

(1 3 5)

Example

Write a function which takes a list of strings an input and outputs those strings which starts with 's'.

Step1: Write a function which takes a string as input and output true if it starts with 's' else f.

(: starts-with-S? : String -> Boolean)
(define (starts-with-S? title)

```
(and (> (string-length title) 0)

(char=? (string-ref title 0)
        #\S ))))
```

```
(: list-starts-S : (Listof String)
   → (Listof String))
(define (list-starts my-list)
(filter starts-with-S? my-list))
```

Function which checks if a string starts with S

Input

Foldr and Foldl

(foldr + 0 (list 1 2 3 4))

Acc = 0 ⟹ 10

(Binary operator → A function which take two inputs of the same type and outputs something of the same type)

4 + Acc = 4

Acc = Acc + 3

Acc = Acc + 2

Acc = Acc + 1

Before val

Acc = 0

Acc = 4

Acc = 7

Acc = 9

Acc = 10.

Write a program which takes a list of integers an input and sums all its elements

```
(: sum-list : (Listof Integer) -> Integer)
(define (sum-list my-list)
  (foldr + 0 my-list))
```

```
(foldl + 0 (list 1 2 3 4))
```

⇒ 10.

The ↓ string-append is the binary operator

```
(foldr string-append "" (list "h" "e" "l" "l" "o"))
```

⇒

Step1: Element from the Acc

string-append "o" "" ⇒ "o"

string-append "l" "o" ⇒ "lo"

string-append "l" "lo" ⇒ "llo"

string-append "e" "llo" ⇒ "ello"

string-append "h" "ello" ⇒ "hello"

```
(foldl string-append "" (list "h" "e" "l" "l" "o"))
```

⇒ "olleh"

Polymorphism

There is function called length.

```
(length (list 5 10 100))
```

⇒ 3

```
(length (list "Hyde" "Pork"))
```

$\Rightarrow$ 2.

length works list of any type.

polymorphism - polymorphic function

(: length : (All (A) (Listof A) → Integer))

First

What is the type annotation for first?

(: first : (All (A) (Listof A) → A))

(: rest : (All (A) (List of A)
→ (List of A)))

What is the type of map

Input 1: Function (A → B)
Input 2: List of A.

Output: List of B.

(: map : (All (A B) (A → B)
(List of A) → (List of B)))

Type annotations for foldl and foldr

First-Class function

Function are Data

Example of a function which outputs a function.

Example
Write a function which takes two functions and a real number as input, the function which was greater value at the that real number.

Input: f, g, t

$f(t) > g(t)$

$\Rightarrow f$

otherwise

$\Rightarrow g$

```
(: max-func : (Real -> Real)
   (Real -> Real) Real
   -> (Real -> Real))

(define (max-func f g z)
  (if (> (f z) (g z))
      f
      g))
```

$f(x) = x + 1$

$g(x) = x^3$

Inputs $f(x) = x+1$

$g(x) = x^3$

2

Output: $f(2) = 3$

$g(2) = 8$

$\Rightarrow g$

```
(: add-one : Real → Real)
(define (add-one x)
  (+ x 1))

(: cube : Real → Real)
(define (cube x)
  (* x x x))

(check-expect ((max-func-add-one cube 2) 5)
              (cube 5))
```

Example

```
(define-type Int-Set (Integers
                      → Boolean))
```

Concept of set of integers.

Set of Integers : $\mathbb{Z}$

$\{2, 3\} \subseteq \mathbb{Z}$

$\{1, 5, 6, 7\} \subseteq \mathbb{Z}$

$\{2, 4, 6, 8, 10, 12 \ldots\} \subseteq \mathbb{Z}$

The set of all even integers

A set of integers can though of as a function.

It's function from Integers to Boolean

$\{2, 3\} \subseteq \mathbb{Z}$.

$f(2) \Rightarrow$ true

$f(3) \Rightarrow$ true

$f(n) \Rightarrow$ false
for any other integer n.

$(f\ 4) \Rightarrow$ #false

Function representation of sets.

Want to represent the set $\{2, 3\}$

```
(: 2-or-3 : Int-Set)
;; This function represents
   the set {2 3}
(define (2-or-3 n)
  (match n
    [2  #t] ← True
    [3  #t] ← True
    [_  #f] ← False.
```

2-or-3 "represents" the set {2, 3}

Example2:
Write a function which takes two Int-Set (two sets) outputs the union.

$\{2,3\} \cup \{3,4,5\}$
$\Longrightarrow \{2,3,4,5\}$

Ion, other words.
f1 = {2, 3}
f2 = {3, 4, 5}

Then my output should be f3 which outputs true on {2,3,4,5} else false.

```
(: union-int-set : Int-Set
   Int-Set → Int → Set)
;; takes the union of
;; two int-sets
```

I am defining the set.

```
(define (union-int-set set1 set2)
  (local ← This lets define things inside a function
    {
     (: set3 : Int → Set)
     (define (set3 n)
       (or (set1 n) (set2 n)))}
    set3))
```

set1 and set2 are available here.

```
(define (S n)
 (or (set1 n) (set2 n)))
```

Homework 4 2c

Mutual Recursion

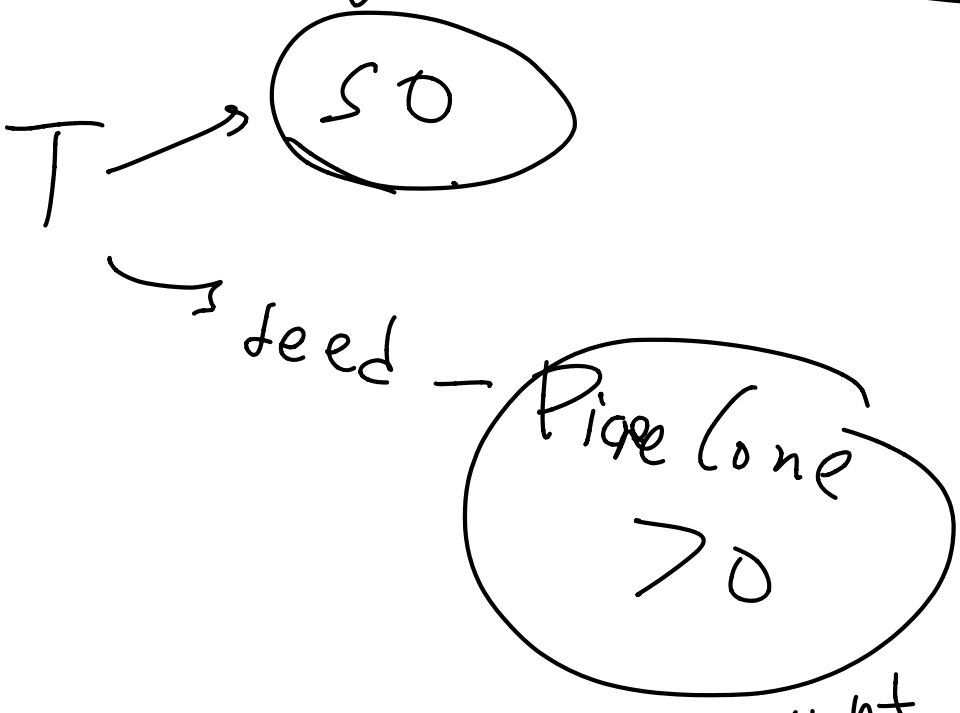
Suppose there is a tree T

T — height

— seed

max-tree-descendant(T)
= max { height , max-cone-descendant
-height(seed) }

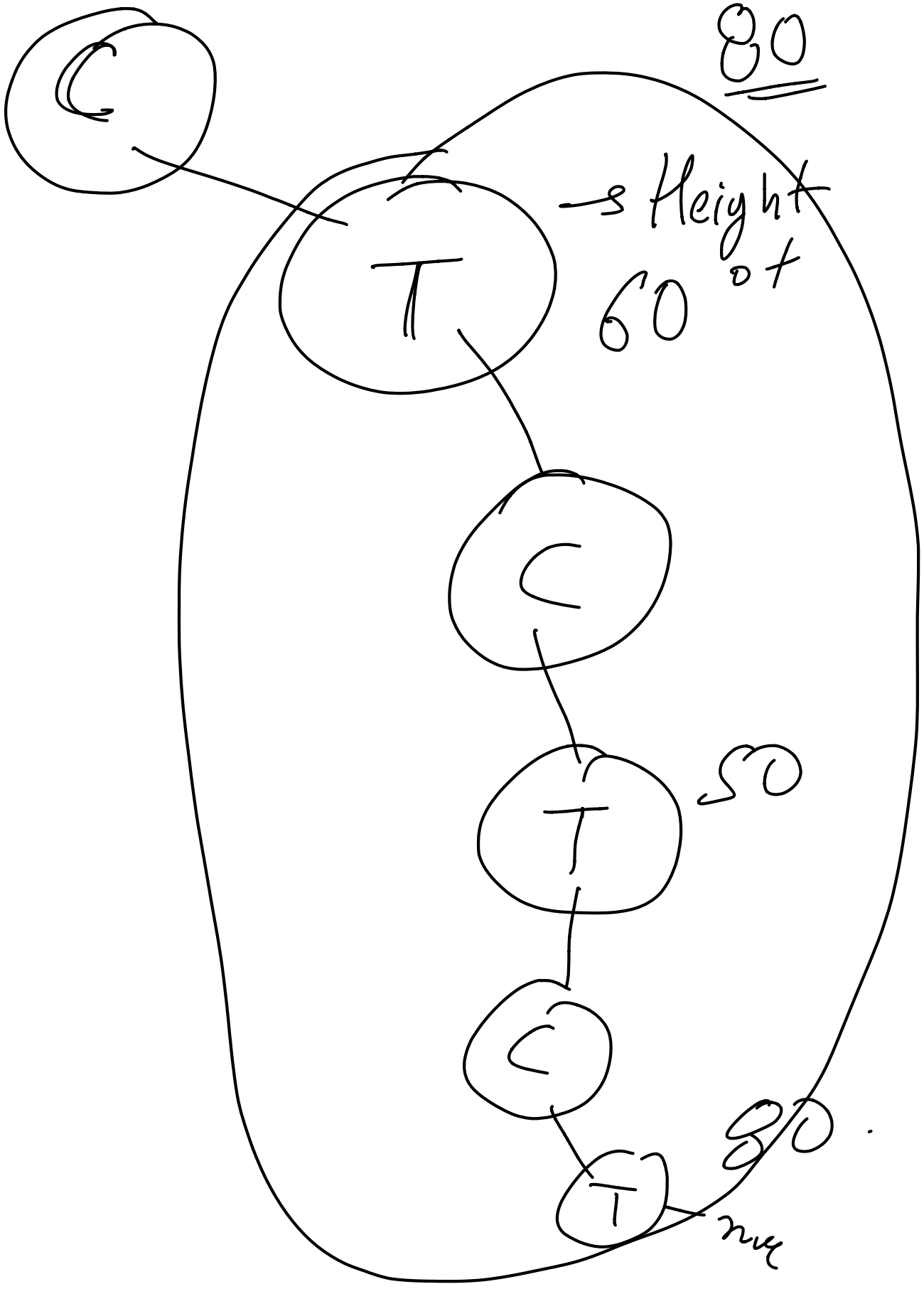
height of T.



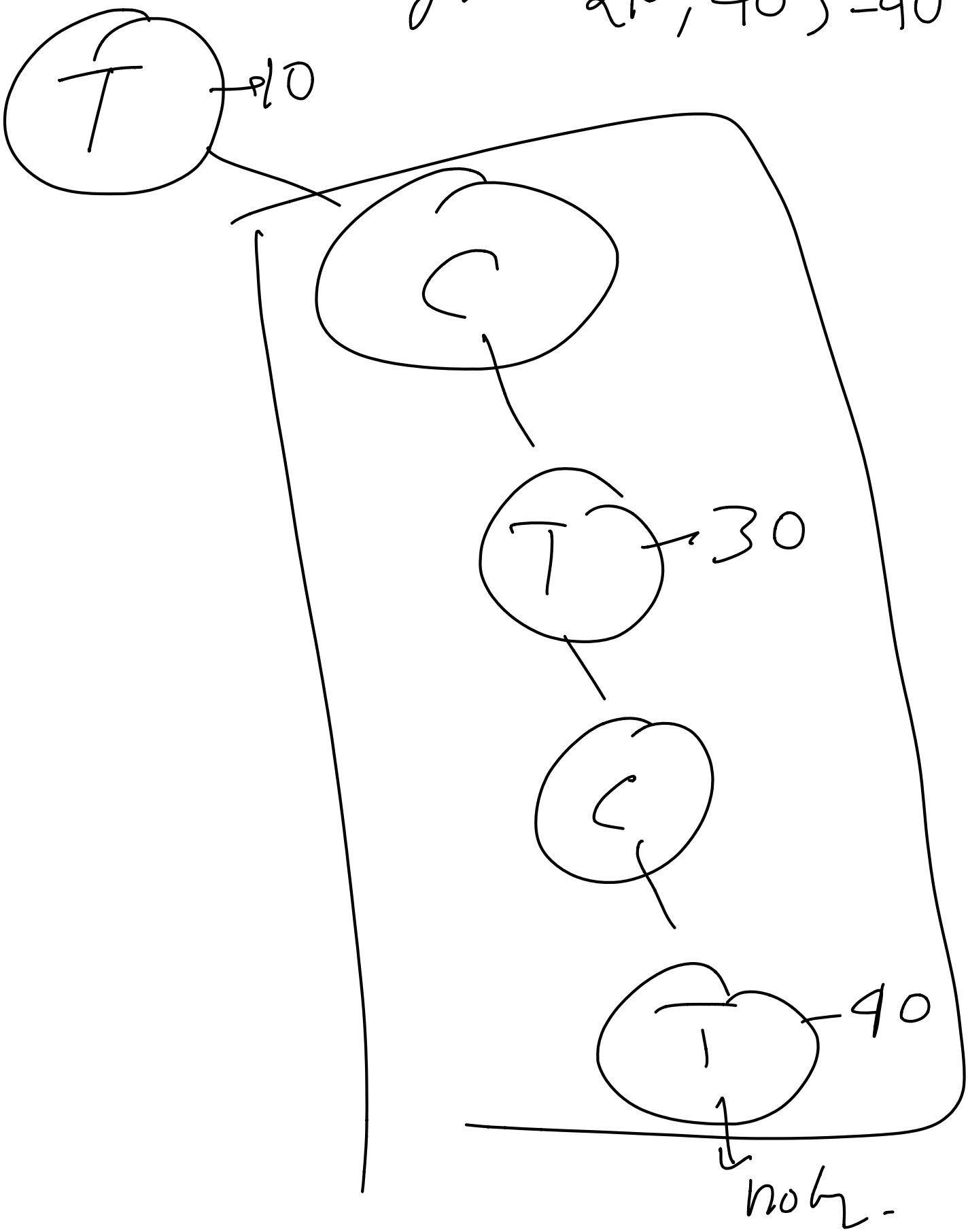
max-cone-descendant(C)
= max-tree-descendant(T)

C is a cone here

C grows in T.

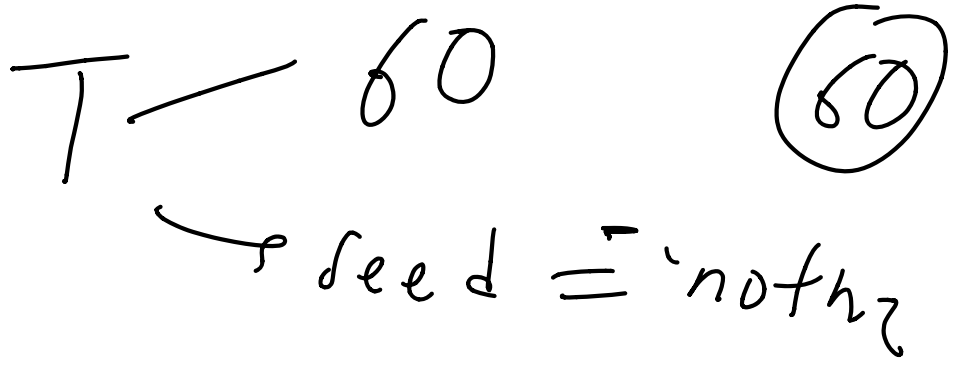


max height = {10, 40} = 40



Base case for
max-tree-descendant
-height

If tree has no seeds.



Base case for max-cone
-descendant-height

If my cone grows into
nothing, its height is 0.

```
(: max-tree-descendant-height
   : Pine-Tree → Real)
(define (max-tree-descendant-height
          tree)
  (cond
[(symbol? (Pine-Tree-seed tree))
  (Pine-Tree-height tree)]

[else (max (Pine-Tree-height tree)

    (max-cone-descendant-height
      (Pine-Tree-seed tree)))]))
```

```
(: max-cone-descendant-height :
   Pine-Cone -> Real)
(define (max-cone-descendant-height
   cone)
(cond
  [(symbol? (Pine-Cone-grows-into
                cone)) 0]
[else (max-tree-descendant-height
        (Pine-Cone-grows-into cone))])
```

2b

```
(: germinate : Pine-Tree -> Pine-Tree)
   (define (germinate tree)
(cond
  [(Pine-Cone? (Pine-Tree-seed tree))
```

```
        tree ]
[else (Pine-Tree (Pine-Cone
  'nothing 40) (Pine-Tree-height
                tree )))))
```