Contents

1 Conceptual T/F

For each of the following tasks, indicate whether the statement is true or false.

1.1 Basics

Task 1.1.

Expressions that are not well-typed will not be evaluated.

Solution 1.

True. Typechecking happens before evaluation.

Task 1.2.

If an expression is well-typed, then it is valuable.

Solution 2.

False. Well-typed expressions are evaluated, but they may not necessarily reduce to a value. Consider `1 div 0`. This has type `int` because `div` expects two expressions of type `int` and has output type `int`, but it raises an exception rather than evaluating to a value.

Task 1.3.

If `e1` $\Longrightarrow$ `e2`, then `e1` $\cong$ `e2`.

Solution 3.

True.

Task 1.4.

If `e1` $\cong$ `e2`, then `e1` $\Longrightarrow$ `e2`.

Solution 4.

False. $\Longrightarrow$ is stronger than $\cong$. We say that `e` $\Longrightarrow$ `e'` if SML would reduce `e` to `e'` (ex: forward stepping through code). For expressions `e : t` and `e' : t` (where `t` is a non-function type), we say that `e` $\cong$ `e'` if both expressions either:

- reduce to the same value,
- raise the same exception, or
- loop forever.

For example, `3 + 4` $\cong$ `2 + 5` because both expressions reduce to `7`, but we cannot say that `3 + 4` $\Longrightarrow$ `2 + 5` because SML does not make this step.

Remark: Extensional equivalence is defined slightly differently for functions. Suppose `f : t1 -> t2` and `f' : t1 -> t2`. Then we say that `f` $\cong$ `f'` if for all values `v : t1`, `f v` $\cong$ `f' v`.

Task 1.5.

SML evaluates the arguments to a function before stepping through the body of the function.

Solution 5.

True. SML is an eagerly evaluated language.

Task 1.6.

If a function `f : t1 -> t2` is valuable, then it is total.

Solution 6.

False. Consider `fact : int -> int`. `fact` is a value (because function values are values), but `fact` loops forever on negative `int`s.

Task 1.7.

`case (3 + 1) of (2 + 2) => "four" | _ => "not four"` reduces to `"four"`.

Solution 7.

False. If we tried evaluating the LHS in the REPL, it would give us an error. Non-value expressions are not valid patterns, so we would not be able to use `2 + 2` as a pattern.

Task 1.8.

`fn (x : int) : int => x + 1 + 1` reduces to `fn (x : int) : int => x + 2`.

Solution 8.

False. Lambda expressions are values, so they do not reduce further. In particular, we do not evaluate the body of the function unless the arguments are passed in.

Task 1.9.

The following declares a recursive function `foo : int -> int`:

```
val foo = (fn (0 : int) => 0 | n => 1 + foo (n - 1))
```

Solution 9.

False. `val` declarations reduce the expression down to a value, then bind the variable to that value. In the lambda expression, `foo` would be an unbound variable because the `val` binding for `foo` has not been made yet.

We can use `fun` declarations instead for recursive functions.

1.2 Induction/Recursion

Task 1.10.

Consider the following declaration for `foo : int -> int`. If we want to prove a theorem about `foo n` for all positive `n : int`, we can use simple/weak induction to do so.

```
fun foo (1 : int) : int = 1
  | foo (2 : int) : int = 3
  | foo (n : int) : int = foo (n - 1) + foo (n - 2)
```

Solution 10.

False. To prove that the theorem holds for some $n > 2$, we would have to know something about both `foo (n - 1)` and `foo (n - 2)`. With simple/weak induction, we would only be able to make one assumption. Instead, we could use strong induction, which assumes the statement holds for all `k : int` where $1 \leq k < n$.

Task 1.11.

For which values of (A) and (B) is `f 150` valuable?

```
fun f ((A) : int) : int = 0
  | f ((B) : int) : int = 1
  | f (n : int) : int = f (n - 1) + f (n - 2)
```

(a) `(A)` is `0`, `(B)` is `1`.

(b) `(A)` is `0`, `(B)` is `2`.

(c) `(A)` is `1`, `(B)` is `2`.

(d) `(A)` is `1`, `(B)` is `50`.

Solution 11.

(a) and (c). If both of our base cases are less than 150, then they should be consecutive. (Looking at it inductively, to show that `f n` is valuable in the inductive step, we would need to assume that `f (n - 1)` and `f (n - 2)`) are valuable. If the base cases are not consecutive, then we would not be able to use the inductive step for any values because we do not have both assumptions.)

Task 1.12.

Any statement that can be proven by weak induction can also be proven by strong induction.

Solution 12.

True. Strong induction makes more assumptions than weak induction with the IH (specifically, that the statement holds for all smaller cases).

For Tasks 1.13 and 1.14, suppose we are given the implentations of two functions `f : int -> t` and `g : int -> t`. We want to show that `f x` $\cong$ `g x` for all values `x` $\geq$ `0` (where `x : int`). (If a task is false, provide a (small) correction to the IH that could potentially make it work.)

Task 1.13.

The following could be a valid inductive hypothesis in an induction proof:

IH: Assume that for all values `x : int`, we have `f x` $\cong$ `g x`.

Solution 13.

False. We do not want to assume the entire theorem in our IH. (In this case, we would also be assuming the theorem holds for negative `x`, which may not be true.)

One possible fix:

IH: Assume that for ~~all values~~ **some value** `x : int`, we have `f x` $\cong$ `g x`.

We would also quantify `x` accordingly (ex: `x` $\geq$ `0`).

Task 1.14.

In a strong induction proof, our inductive hypothesis does not have to cover the values of any of our base cases. For example, the quantification for `y` in the inductive hypothesis below is correct.

BC 1: `x` $=$ `0`. (proof omitted)

BC 2: `x` $=$ `1`. (proof omitted)

IS: `x` $>$ `1`.

IH: Assume that for all values `y : int` satisfying `2` $\leq$ `y` $<$ `x`, we have `f y` $\cong$ `g y`.

Solution 14.

False. Because we are citing the IH in our inductive step, we would want to include the appropriate base case values.

One possible fix:

IH: Assume that for all values `y : int` satisfying $\cancel{2 \leq y < x}$ `0` $\leq$ `y` $<$ `x`, we have `f y` $\cong$ `g y`.

Task 1.15.

Implement the tail-recursive helper function `max'` such that `max` satisfies the following specs:

`max : int list -> int option`

REQUIRES: true

ENSURES: `max L` $\Longrightarrow \begin{cases} \texttt{SOME v} & \text{if } \texttt{v} \text{ is the max element of } \texttt{L} \\ \texttt{NONE} & \text{if } \texttt{L} \text{ is empty.} \end{cases}$

```
fun max' (L : int list, acc : int option) : int option =
    raise Fail "your implementation here"

fun max (L : int list) : int option = max' (L, NONE)
```

Solution 15.

```
fun max' ([] : int list, acc : int option) : int option = acc
  | max' (x::xs, NONE) = max' (xs, SOME x)
  | max' (x::xs, SOME y) =
      if x < y
      then max' (xs, SOME y)
      else max' (xs, SOME x)
```

1.3 Lists/Datatypes

Task 1.16.

`1::2::3::[]` is a value.

Solution 16.

True. Constructors of a datatype are used to create values of that type. (Note that `[1, 2, 3]` is syntactic sugar for `1::2::3::[]`, and since the latter is a value, we would not say that `1::2::3::[]` reduces to `[1, 2, 3]`.)

Task 1.17.

The `type` and `datatype` keywords are interchangeable (i.e. `type` and `datatype` declarations are the same).

Solution 17.

False. We use `datatype` declarations to define value constructors of a type, whereas `type` declarations are used to give another name to an already existing type. For example:

```
type intPair = int * int
datatype pearTree = Pear of intPair
```

```
| Branch of pearTree * pearTree
```

Task 1.18.

`Node` has type `tree`, where the `datatype` declaration for `tree` is as follows:

```
datatype tree = Empty | Node of tree * int * tree
```

Solution 18.

False. The type of `Node` is `tree * int * tree -> tree` since the `Node` constructor takes in an argument of type `tree * int * tree`.

Remark: Although `Node` has a function type, constructors are not exactly functions. Here are a couple of differences between the two:

- For a function value `f : t1 -> t2` and a value `v : t1`, `f v` can be reduced. However, for a constructor `C : t1 -> t2` and a value `v : t1`, `C v` is already a value and cannot be reduced further.
- We can use `C x` as a pattern for some pattern `x`, but `f x` cannot be used as a pattern.

Task 1.19.

Given the following `datatype` declaration for `varTree`, `Two (One x, _)` is a valid pattern.

```
datatype varTree = Zero | One of varTree | Two of varTree * varTree
```

Solution 19.

True. Constructors, tuples of patterns, and wildcards are all patterns.

Task 1.20.

With structural induction on `trees`, we induct directly on the number of nodes in the `tree`.

Solution 20.

False. In structural induction, we induct directly on the type's structure. Each constructor would either correspond to a base case or an inductive case. For `trees`, `Empty` is a base case because it is a non-recursive constructor, and `Node (L, x, R)` is an inductive case because it is a recursive constructor.

1.4 Work/Span

Task 1.21.

The following implementation of `rev : int list -> int list` has $O(n)$ work, where n is the length of the input list.

```
fun rev ([] : int list) : int list = []
  | rev (x::xs) = (rev xs) @ [x]
```

Solution 21.

False. This has $O(n^2)$ work. The following implementation calls a tail-recursive helper that takes in an accumulator and has $O(n)$ work.

```
fun rev' ([] : int list, acc : int list) : int list = acc
  | rev' (x::xs, acc) = rev' (xs, x::acc)

fun rev (L : int list) : int list = rev' (L, [])
```

Task 1.22.

The best-case work for calling `inord` is when the input tree is balanced.

```
fun inord (Empty : tree) : int list = []
  | inord (Node (L, x, R)) = (inord L) @ (x :: (inord R))
```

Solution 22.

False. This has $O(n \log n)$ work, where n is the number of nodes in the tree. The best-case would be when all nodes are along the right spine, a maximally unbalanced tree, which has $O(n)$ work.

Remark: The best-case for a tree is not always a balanced tree. Similarly, the worst-case is not always a maximally unbalanced tree. When trying to figure out the best/worst-case, write out the recurrences, using n_l and n_r to denote the size of the left and right subtrees, respectively. Then, use the non-recursive work/span per node and the number of recursive calls to the subtrees to reason through what the best/worst-case may look like. (In some cases, the work/span will only depend on the number of nodes, and the structure of the tree will not matter.)

Task 1.23.

When calculating span, we assume that expressions in tuples are evaluated in parallel.

Solution 23.

True. (This is our parallelism policy.)

Task 1.24.

When calculating span, we assume that `val` declarations in `let...in...end` expressions are executed in parallel.

Solution 24.

False. These are executed sequentially. Given a sequence of `val` declarations, it is possible for some of them to use bindings made earlier on, so they may not be independent.

Task 1.25.

Suppose the recursive function `foo` has input type `tree`. In the tree method, the work tree for `foo` always has 2^i nodes at level i when the input tree is balanced.

Solution 25.

False. The work tree is not the same as the input tree. Consider the following example:

```
fun foo (Empty : tree) : int = 0
  | foo (Node (L, x, R)) = 1 + foo L + (foo R + foo R) div 2
```

This would correspond to the following recurrences:

$$W_{\texttt{foo}}(0) = c_0$$
$$W_{\texttt{foo}}(n) = 3W_{\texttt{foo}}(\tfrac{n}{2}) + c_1$$

At level i, the work tree would have 3^i nodes.

Task 1.26.

If the work of a function is in $O(f(n))$, then its span is also in $O(f(n))$.

Solution 26.

True. However, note that we usually ask for tight bounds, so if there is room for parallelism, the asymptotic work and span bounds may differ.

2 Types and Evaluation

Assume the following is in scope:

```
datatype tree = Empty | Node of tree * int * tree
```

For each of the following declarations:

- If it typechecks, give the type of `v`; otherwise state "not well typed." and explain why it has no type.
- If `v` is valuable, give its value. Otherwise, give "no value." and explain why it has no value.

Task 2.1. (Recommended)

```
val v = 3 / 2
```

Solution 1.

Type: not well-typed since `/` is used to divide `reals`
Value: no value since it is not well-typed

Task 2.2.

```
val v = 3 + 2.0
```

Solution 2.

Type: not well-typed since `+ : int * int -> int` or `+ : real * real -> real`, but no other combination
Value: no value since ill-typed

Task 2.3. (Recommended)

```
val v = 1 div 0
```

Solution 3.

Type: `int`
Value: no value (raises `Div` exception)

Task 2.4. (Recommended)

```
val v = fn x : int => x + 1
```

Solution 4.

Type: `int -> int`
Value: `fn x => x + 1`

Task 2.5.

```
val v = fn (x, y) => x andalso y = 3
```

Solution 5.

Type: `bool * int -> bool`
Value: `fn (x, y) => x andalso y = 3`

Task 2.6. (Recommended)

```
val v = fn (x : int) => Node (Empty, x, Empty)
```

Solution 6.

Type: `int -> tree`
Value: `fn x => Node(Empty, x, Empty)`

Task 2.7.

```
val v = ()
```

Solution 7.

Type: `unit`
Value: `()`

Task 2.8. (Recommended)

```
val v = [[1]] @ []
```

Solution 8.

Type: `int list list`
Value: `[[1]]`

Task 2.9.

```
val v = [150] :: []
```

Solution 9.

Type: `int list list`
Value: `[[150]]`

Task 2.10.

```
val v = fn (x : int) => x :: [1]
```

Solution 10.

Type: `int -> int list`
Value: `fn x => x :: [1]`

Task 2.11. (Recommended)

```
val v = (fn (x : int list) => x :: []) [1]
```

Solution 11.

Type: `int list list`
Value: `[[1]]`

Task 2.12.

```
fun v (x : int) : int = x = 3
```

Solution 12.

Type: Not well-typed, `x = 3 : bool`, but the annotated return type of `v` is `int`
Value: No value since not well-typed

Task 2.13. (Recommended)

```
val v = if 5 > 0 then "polly" else 150
```

Solution 13.

Type: not well-typed, both clauses need to have the same type
Value: no value since not well-typed

Task 2.14.

```
fun v (Empty) = []
  | v (Node (l, x, r)) = v l @ x @ v r
```

Solution 14.

Type: not well typed. `x` must have type `int list` for `@ : int list * int list -> int list`, but `x` has type `int`
Value: no value since not well typed

Task 2.15. (Recommended)

```
fun v (a, []) = ([a], a + 1)
  | v (_, x :: xs) = v (not x, xs)
```

Solution 15.

Type: not well-typed (by type inference, the first argument to `v` should have type `int`, but `not x` has type `bool`)
Value: no value since ill-typed

Task 2.16. (Recommended)

```
val v =
  let
    fun g [] = g (["150"])
      | g (x :: xs) = x ^ g xs
  in
    (g, g [], g ["3"])
  end
```

Solution 16.

Type: `(string list -> string) * string * string`
Value: no value, `g []` will loop forever (as will `g ["3"]`)

Task 2.17.

```
fun v (b : bool) =
let
    val f = fn x => x
in
    (f 10, f b)
end
```

Solution 17.

Type: `bool -> int * bool`
Value: `fn b => let val f = fn x => x in (f 10, f b) end`

3 Elections (Datatypes)

You are working as a functional programmer for SML mega-corp Church Inc. You learn that Church Inc.'s main rival, Turing Farms, has moved into town. The town council has decided that there is only room for one of the two, and it is up to a vote to decide who stays. As one of their most trusted SML programmers, Church Inc. has asked you to help them get ahead in the election. They have a slew of SML functions that they need you to write so that they can understand what might happen in the election.

We first define

```
datatype vote = A | B
type election = vote list
```

Note that a `A` vote represents a vote for Church Inc. and a vote for `B` represents a vote for Turing Farms.

To start, we need to figure out the results of a given election.

For this task, we will be working in `code/elections/elections.sml`.

Task 3.1.

Implement the following function

`result : election -> int`

REQUIRES: true

ENSURES: `result e` $\Longrightarrow$ `s`, the sum of votes for `A` in `e` - sum of votes for `B` in `e`

Solution 1.

```
fun result (e : election) : int =
  (case e of
    [] => 0
  | A::xs => 1 + result xs
  | B::xs => ~1 + result xs)


(* Test cases *)
(*
val () = Test.int("result_empty", 0, result [])
val () = Test.int("result_A", 1, result [A])
val () = Test.int("result_B", ~1, result [B])
val () = Test.int("result_AB", 0, result [A,B])
val () = Test.int("result_AA", 2, result [A,A])
val () = Test.int("result_BBA", ~1, result [B,B,A])
*)
```

Next, Church Inc. wants to know whether or not they were winning the whole time for some given election.

Task 3.2. (Recommended)

Implement the following function

`always_winning : election -> bool`

REQUIRES: `result e` > 0

ENSURES: `always_winning(e)` $\Longrightarrow$ `true` if at all points when counting the votes from left to right, there are *strictly more* votes for `A` than there are for `B`

Hint: Defining a helper function might be a nice idea.

Solution 2.

```
fun always_winning (e : election) : bool =
  let
    fun always_winning_helper (e : election, n : int) : bool =
      (case (e, n <= 0) of
        (_, true) => false
      | ([], _) => true
      | (A::xs, _) => always_winning_helper (xs, n + 1)
      | (X::xs, _) => always_winning_helper (xs, n - 1))
  in
    (case e of
      B::xs => false
    | A::xs => always_winning_helper (xs,1)
    | _ => raise Fail "illegal input")
  end

(* Test cases *)
(*
val () = Test.bool("always_winning_A", true, always_winning [A])
val () = Test.bool("always_winning_BAA", false, always_winning [B,A,A])
val () = Test.bool("always_winning_AABAB", true, always_winning [A,A,B,A,B])
*)
```

Oh no! After creating a list of possible elections, Church Inc. realized that they forgot to include your vote[a]! They want you to prepend your vote to each of the elections in the list they created. Being clever, you decide to do it with the following SML function.

[a] Don't worry if you're not 18. In this world, all things (even functions) are first-class citizens, and thus entitled to a vote!

Task 3.3. (Recommended)

Define the following function

```
election_map : election list * vote -> election list
```

REQUIRES: true

ENSURES: `election_map (E,v)` $\Longrightarrow$ `L`, an election list that is the same as `E` but has the vote `v` prepended to all elections in `E`

Solution 3.

```
fun election_map (E : election list, v : vote) : election list =
  (case E of
    [] => []
  | x::xs => (v::x)::election_map(xs, v))

(* Test cases *)
(*
val () = Test.general_eq("election_map_1", [], election_map([], A))
val () = Test.general_eq("election_map_2", [[A]], election_map([[]], A))
val () = Test.general_eq("election_map_3", [[B,A],[B],[B,A,A]],
                                          election_map([[A],[],[A,A]], B))
*)
```

Not sure that their preparation was enough, Church Inc. wants to look at all possible elections with n votes.

Task 3.4. (Recommended)

Define the following function

```
all_elections : int -> election list
```

REQUIRES: `n` ≥ 0

ENSURES: `all_elections n` $\Longrightarrow$ `L`, a list of all elections with `n` votes

Hint: `election_map` might be helpful

Here are some possible outputs:

```
all_elections 2 = [[A, A], [A, B], [B, A], [B, B]]
all_elections 0 = [[]]
```

Solution 4.

```
fun all_elections (n : int) : election list =
  (case n of
    0 => [[]]
  | _ => let
           val other_elections = all_elections (n - 1)
         in
           election_map (other_elections, A) @ election_map (other_elections, B)
```

```
        end)

(* Test cases *)
(*
val () = Test.general_eq("all_elections_0", [[]], all_elections 0)
val () = Test.general_eq("all_elections_1", [[A],[B]], all_elections 1)
val () = Test.general_eq("all_elections_2", [[A,A],[A,B],[B,A],[B,B]],
    all_elections 2)
val () = Test.general_eq("all_elections_3", [[A,A,A],[A,A,B],[A,B,A],[A,B,B],
    [B,A,A],[B,A,B],[B,B,A],[B,B,B]], all_elections 3)
*)
```

After doing more research, Church Inc. is pretty sure they know exactly how many votes both they and Turing Farms will get. They want to see all possible elections where this happens.

Task 3.5.

Implement the following function

`all_perms : int * int -> election list`

REQUIRES: $\texttt{a}, \texttt{b} \geq 0$

ENSURES: `all_perms (a,b)` $\Longrightarrow$ `L`, a list of all elections where `A` receives `a` votes and `B` receives `b` votes

Hint: Try filtering `all_elections` by creating a helper function.

Hint: You may find the `result` function from earlier helpful!

Solution 5.

```
fun all_perms (a : int, b : int) : election list =
  let
    val all = all_elections (a + b)
    fun election_filter (E : election list) : election list =
      (case E of
        [] => []
      | x::xs => if result x = a - b
                 then x :: election_filter xs
                 else election_filter xs)
  in
    election_filter all
  end

(* Test cases *)
(*
val () = Test.general_eq("all_perms_00", [[]], all_perms (0,0))
val () = Test.general_eq("all_perms_10", [[A]], all_perms (1,0))
val () = Test.general_eq("all_perms_11", [[A,B],[B,A]], all_perms (1,1))
val () = Test.general_eq("all_perms_12", [[A,B,B],[B,A,B],[B,B,A]], all_perms
(1,2))
val () = Test.general_eq("all_perms_40", [[A,A,A,A]], all_perms (4,0))
*)
```

While they are pretty sure they are going to win, Church Inc. wants to be extra confident of this. They want to see all of the elections where they receive a votes, Turing Farms receives b votes, and Church Inc. is winning the whole time. (Votes are counted from left to right; winning means having strictly more votes than your opponent.)

Task 3.6.

Implement the following function

```
winning : int * int -> election list
```

REQUIRES: $a > b$

ENSURES: `winning (a,b)` $\Longrightarrow$ `L`, a list of all elections where `A` wins by receiving `a` votes and `B` receives `b` votes and there are always more votes for `A` than there are for `B`

Solution 6.

```
fun winning (a : int, b : int) : election list =
  let
    val all = all_perms (a,b)
    fun election_filter (E : election list) : election list =
      (case E of
        [] => []
      | x::xs => if always_winning x
                 then x :: election_filter xs
                 else election_filter xs)
  in
    election_filter all
  end

(* Test cases *)
(*
val () = Test.general_eq("winning_10", [[A]], winning (1,0))
val () = Test.general_eq("winning_40", [[A,A,A,A]], winning (4,0))
val () = Test.general_eq("winning_21", [[A,A,B]], winning (2,1))
*)
```

Church Inc. is extremely thankful to you for showing them all of the ways that the election could play out. To be honest, however, they're actually a little disappointed, after looking at all the possible elections that Turing Farms could win. In a moment of desparation, they find out that they can use Polly to skew the results of the election by adding votes to their favor! The catch? Polly could also subtract votes. With your help, Church Inc. wants to see how Polly's skew will affect the election.

We define the datatype `winner` to represent the winning candidate.

```
datatype winner = AWin | BWin | TIE
```

Polly will first decide if she's up to skewing the election or not, and if she is, she will define a function corresponding to how the result[a] of the election should be skewed.

[a] Recall how the function `result` works: it takes in an election and returns the number of votes for `A` minus the number of votes for `B`.

We define

```
datatype skew = Skew of int -> int | NoSkew
```

where `Skew` contains a function that takes in the result of an election and returns the skewed result.

Task 3.7.

Implement the following function

`skewedResult : election * skew -> winner`

REQUIRES: true

ENSURES: `skewedResult (e, s)` $\Longrightarrow$ `w`, the winning candidate of the election with the skew

Solution 7.

```
fun skewedResult (e : election, s : skew) : winner =
  let
    val electionRes : int = result e
    val finalRes : int = (case s of
                             Skew bias => bias electionRes
                           | NoSkew => electionRes)
  in
    (case (finalRes, finalRes < 0) of
          (0, _) => TIE
        | (_, true) => BWin
        | (_, false) => AWin)
  end
```

Hint: You may find the `result` function from earlier helpful!

4 Totally Tubular Totality Testimonies

Church Inc. has had enough of Turing Farms' malicious schemes, and decided to finally settle things in court! It's up to you to `sort` between the truths and the lies presented by Turing Farms' witnesses.

4.1 Non-Function Types

Let `t1`, `t2`, `t3` be non-function types (e.g. `string`, `int list`, but not `int -> int` or `int list -> int`).

State whether each of the following statements are **true or false**.

- If you say a statement is true, give a justification.
- If you say a statement is false, give a counterexample.

For some of these problems, you will need to recall the `fact` function:

```
fun fact (0 : int) : int = 1
  | fact n = n * fact (n - 1)
```

Furthermore, consider the following `tree` datatype and functions that act on them:

```
datatype tree = Empty | Node of tree * int * tree

fun createTree (0 : int) : tree = Node (Empty, ~1, Empty)
  | createTree x =
      Node (createTree (x - 1), (x - 1), createTree (x - 1))

fun treeMap (f : int -> int, Empty : tree) = Empty
  | treeMap (f, Node (L, x, R)) =
      Node (treeMap (f, L), f x, treeMap (f, R))
```

Task 4.1.

`fact x` is valuable for all non-negative values `x : int`.

Solution 1.

True - `fact x` will always evaluate to a value so long as `x` is a non-negative value.

Task 4.2. (Recommended)

`createTree x` is valuable for all non-negative values `x : int`.

Solution 2.

True - as long as `x : int` is a non-negative value, `createTree` will always return a value. We can also prove the valuability of `createTree x` for all non-negative values `x : int` via simple induction on `x`.

Task 4.3. (Recommended)

`treeMap (fact, createTree x)` is valuable for all non-negative values `x : int`.

Solution 3.

False - consider `createTree 0`, which returns `Node (Empty, ~1, Empty)`. Then calling `treeMap (fact, createTree 0)` will never terminate as `fact` is called on an input of `~1`.

Task 4.4. (Recommended)

Let `f : int -> int` be total. Then `treeMap (f, T)` is valuable for all values `T : tree`.

Solution 4.

True - this can be proven via structural induction on `T`, using the fact that `f` is total.

Task 4.5.

Let `f : t1 -> (int -> int) = fn x => fact`. Then `f x` is total for all values `x : t1`.

Solution 5.

False - we know that `fact` is not total, which is what `f x` returns.

Task 4.6.

Let `f : t1 -> (t2 -> t3)` be total. Then `f x` is valuable for all values `x : t1`.

Solution 6.

True - `f` is total, so by definition `f x` is valuable for all values `x`.

Task 4.7. (Recommended)

Let `f : t2 -> t2 = fn x => x` and `g : t1 -> t2`. Then we can step `f (g x)` $\Longrightarrow$ `g x` so long as `x : t1`.

Solution 7.

False - They are extensionally equivalent, but this step is not actually accurate. This is since `g x` will be evaluated to some value `v` first.

Task 4.8.

Let `f : (t1 -> t2) -> t3` be total. Then `f g` is valuable for all values `g : t1 -> t2`.

Solution 8.

True - this is the definition of totality.

5 Totality

Congratulations!! Due to your hard work and dedication, Church Inc. has won the election! However, more trouble lies ahead.

Tired of writing contracts all day, Turing Farms lawyers have tried a new way of taking Church Inc. down. They have removed all totality citations from Church Inc.'s proofs in an effort to ruin their credibility. Help them out by deciding which of the steps need totality citations and for what functions.

Task 5.1. (Recommended)

For each of the following steps (A - H), add citations for totality (if any) for specific function(s).

Consider the following functions:

```
datatype tree = Empty | Node of tree * int * tree

fun treesum (Empty : tree) : int = 0
  | treesum Node(l, x, r) = x + treesum l + treesum r

fun preorder (Empty : tree) : int list = []
| preorder Node(l, x, r) = x::(preorder l @ preorder r)

fun listsum ([] : int list) : int = 0
  | listsum x::xs = x + listsum xs
```

Lemma 5.1. For all values `l1, l2 : int list`, `listsum (l1@l2)` $\cong$ `listsum l1 + listsum l2`.

Now let's prove the following theorem:

Theorem 5.2. For all values `t : tree`, `treesum t` $\cong$ `listsum (preorder t)`.

Proof. Proof by induction on the structure of trees.

Base Case: Suppose that `t = Empty`. Then:

`treesum Empty` $\cong$ `0`	by clause 1 of `treesum` (A)
`listsum(preorder Empty)` $\cong$ `listsum []`	by clause 1 of `preorder` (B)
$\cong$ `0`	by clause 1 of `listsum` (C)

Induction Hypothesis (IH): Let `l : tree` and `r : tree` be values. Assume that:

`treesum l` $\cong$ `listsum (preorder l)`

`treesum r` $\cong$ `listsum (preorder r)`

Induction Step: We want to show that this holds for `Node(l,x,r)`, where `x` is some `int`.

`listsum (preorder (Node(l, x, r)))`	
$\cong$ `listsum (x::(preorder l @ preorder r))`	by clause 2 of `preorder` (D)
$\cong x +$ `listsum (preorder l @ preorder r)`	by clause 1 of `listsum` (E)
$\cong x +$ `listsum (preorder l) + listsum (preorder r)`	by Lemma 5.1(F)
$\cong x +$ `treesum l + treesum r`	by IH (G)
$\cong$ `treesum (Node(l, x, r))`	by clause 2 of `treesum (H)`

□

Solution 1.

(A) None

(B) None

(C) None

(D) None

(E) Totality of `preorder` and `@`
We are stepping through `listsum` in step E. Since SML is an eagerly evaluated language, all arguments to a function must be evaluated to a value before we can step through the function. Therefore, we need to know that `x :: (preorder l @ preorder r)` is valuable in order to step through `listSum`. First, `preorder l` and `preorder r` are valuable because `preorder` is total. Then, `preorder l @ preorder r` is valuable because `@` is total. Thus `x :: (preorder l @ preorder r)` is valuable.

(F) Totality of `preorder`
Note that the lemma only holds for **values** `l1, l2 : int list`. Since `l1` and `l2` are `preorder l` and `preorder r` for us, we need to cite the totality of `preorder` to show that `preorder l` and `preorder r` are valuable.

(G) None

(H) None

6 Scope (Evaluation)

This question tests some code tracing with scope and binding.

6.1 Scope Training

Task 6.1.

What variables (of what types) must be in scope in order to evaluate this expression?

```
let
    fun f (x : int) = (fn (b : bool) => if b then x else y)
    val g = f z
in
    g (k = w orelse t orelse w = "hello")
end
```

Solution 1.

`y : int`
In the `fun` declaration for `f`, `x` is type-annotated to have type `int`. In order for the expression `if b then x else y` to typecheck, `x` and `y` must have the same type.

`z : int`
The input type of `f` is `int`, so `w` should also have type `int` for `f w` to typecheck.

`k : string`
`w : string`
From the expressions `k = w` and `w = "hello"`, both `k` and `w` are of type `string`.

`t : bool`

6.2 iNoodle

```
val x = 15
val f = (fn x => x + 150)
val g = (fn y => x + y)
val x = f x
val h = (fn z => g x + f z)
val mystery = let val x = 5 in g x end
val meat = let val x = h mystery in f x end
val cases = (case (x,meat) of (f,x) => f + x)
```

Task 6.2.

What is the value that binds to `mystery`?

Solution 2.

20

The environments after each binding are as follows (with the closures (1), (2), (3) indicated below):

```
val x = 15
  [15/x]
val f = (fn x => x + 150)
  [(1)/f, 15/x]
val g = (fn y => x + y)
  [(2)/g, (1)/f, 15/x]
val x = f x
  [165/x, (2)/g, (1)/f]
val h = (fn z => g x + f z)
  [(3)/h, 165/x, (2)/g, (1)/f]
val mystery = let val x = 5 in g x end
  [20/mystery, (3)/h, 165/x, (2)/g, (1)/f]
val meat = let val x = h mystery in f x end
  [500/meat, 20/mystery, (3)/h, 165/x, (2)/g, (1)/f]
val cases = (case (x,meat) of (f,x) => f + x)
  [665/cases, 500/meat, 20/mystery, (3)/h, 165/x, (2)/g, (1)/f]
```

`(1)` is the closure with lambda expression `(fn x => x + 150)` and environment `[15/x]` (note that `x` is shadowed within the lambda expression).

`(2)` is the closure with lambda expression `(fn y => x + y)` and environment `[(1)/f, 15/x]`.

`(3)` is the closure with lambda expression `(fn z => g x + f z)` and environment `[165/x, (2)/g, (1)/f]`.

Task 6.3.

What is the value that binds to `meat`?

Solution 3.

500

Task 6.4.

What is the value that binds to `cases`?

Solution 4.

665

7 Search Sort (Trees, Recursion)

For these tasks, you'll implement `searchsort`, which uses a lower and upper bound to sort a tree into a list.

Let's begin with a helper function, `nextsmallest`, which finds the smallest element in the tree that is greater than the argument, `lo`.

Task 7.1. (Recommended)

In `code/searchsort/searchsort.sml`, write the function

```
nextsmallest : tree * int * int -> int
```

REQUIRES: true

ENSURES: `nextsmallest (T, lo, acc)` $\Longrightarrow$ `lo'` such that `lo'` is the smallest element in `T` satisfying `lo` $<$ `lo'` $<$ `acc`, if such an element exists in `T`. Otherwise, `nextsmallest` returns `acc`.

Solution 1.

```
fun nextsmallest (Empty : tree, lo : int, acc : int) : int = acc
  | nextsmallest (Node (L, x, R) : tree, lo : int, acc : int) : int =
      let
        val lo' =
          if (lo < x) andalso (x < acc) then x else acc
      in
        Int.min (nextsmallest(L, lo, lo'), nextsmallest(R, lo, lo'))
      end
```

Great! Now, let's implement `searchsort`.

Task 7.2. (Recommended)

In `code/searchsort/searchsort.sml`, write the function

```
searchsort : tree * int * int -> int list
```

REQUIRES: The elements in `T` are unique.

ENSURES: `searchsort (T, lo, hi)` $\Longrightarrow$ `L` such that `L` is a sorted list of all elements `x` in `T` for which `lo` $<$ `x` $<$ `hi`.

Solution 2.

```
fun searchsort(T : tree, lo : int, hi : int) : int list =
   (case T of
     Empty => []
   | Node (L, x, R) =>
       let
         val next = nextsmallest (T, lo, hi)
       in
         if next = hi
```

```
          then []
          else next :: searchsort (T, next, hi)
        end)
```

8 Inorder Trees (Trees, Structural Induction, Totality)

Consider the following definitions:

```
datatype tree = Empty | Node of tree * int * tree

fun rev ([] : int list) : int list = []
  | rev (x::xs) = rev(xs) @ [x]

fun [] @ R = R
  | (x::xs) @ R = x :: (xs @ R)

fun revTree (Empty : tree) : tree = Empty
  | revTree (Node(L,x,R)) =
      Node(revTree R, x, revTree L)

fun inord (Empty : tree) : int list = []
  | inord (Node(L,x,R)) =
      (inord L) @ (x::inord R)
```

Task 8.1.

Prove that, for all values `T : tree`,

$$\texttt{rev (inord T)} \cong \texttt{inord (revTree T)}$$

You may find the following lemmas useful:

Lemma 8.1. For all valuable expressions `L1: int list`, `L2: int list`,

`rev (L1 @ L2)` $\cong$ `(rev L2) @ (rev L1)`

Lemma 8.2. `inord` is total

Lemma 8.3. `rev` is total

Lemma 8.4. For all valuable expressions `L1: int list`, `L2: int list`, and all values `x: int`,

$$\texttt{(L1 @ [x]) @ L2} \cong \texttt{L1 @ (x::L2)}$$

Lemma 8.5. `revTree` is total

Solution 1.

We proceed by structural induction on `T : tree`.

Base Case. Take `T = Empty`.

LHS:

$$
\begin{aligned}
& \texttt{rev (inord Empty)} & \\
\cong\ & \texttt{rev []} & (\texttt{inord}\text{ clause 1}) \\
\cong\ & \texttt{[]} & (\texttt{rev}\text{ clause 1})
\end{aligned}
$$

RHS:

$$
\begin{aligned}
& \texttt{inord (revTree Empty)} & \\
\cong\ & \texttt{inord Empty} & (\texttt{revTree}\text{ clause 1}) \\
\cong\ & \texttt{[]} & (\texttt{inord}\text{ clause 1})
\end{aligned}
$$

`[]` ≅ `[]`, so `rev (inord Empty)` ≅ `inord (revTree Empty)`.

Inductive Case. Given some arbitrary values `L, R : tree` and `x : int`, let `T = Node (L, x, R)`.

Inductive Hypothesis . Assume the theorem holds for some values `L, R : tree`, so:

$$\texttt{rev (inord L)} \cong \texttt{inord (revTree L)}$$

$$\texttt{rev (inord R)} \cong \texttt{inord (revTree R)}$$

WTS : `rev (inord (Node (L, x, R)))` ≅ `inord (revTree (Node (L, x, R)))`

$$
\begin{aligned}
& \texttt{rev (inord (Node (L, x, R)))} & \\
\cong\ & \texttt{rev ((inord L) @ (x::(inord R)))} & (\texttt{inord}\text{ clause 2}) \\
\cong\ & \texttt{(rev (x::(inord R))) @ (rev(inord L))} & (\text{Lemma 8.1, Lemma 8.2}) \\
\cong\ & \texttt{((rev (inord R)) @ [x]) @ (rev(inord L))} & \\
& & (\text{Lemma 8.2, }\texttt{rev}\text{ clause 2}) \\
\cong\ & \texttt{(rev (inord R)) @ (x::(rev (inord L)))} & \\
& & (\text{Lemma 8.2, Lemma 8.3, Lemma 8.4}) \\
\cong\ & \texttt{(inord (revTree R)) @ (x::(inord (revTree L)))} & (\text{IH})
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{inord (revTree (Node(L, x, R)))} & \\
\cong\ & \texttt{inord (Node(revTree R, x, revTree L))} & (\texttt{revTree}\text{ clause 2}) \\
\cong\ & \texttt{(inord (revTree R)) @ (x::(inord (revTree L)))} & \\
& & (\text{Lemma 8.5, }\texttt{inord}\text{ clause 2})
\end{aligned}
$$

`rev (inord (Node (L, x, R)))` ≅ `inord (revTree (Node (L, x, R)))`, so we have shown that the theorem holds for this case.

Thus, we have proven the theorem through structural induction on `T`.

9 List-o-mania (Lists, Structural Induction, Work/Span)

Consider the following functions:

```
fun unravel [] = ([], [])
  | unravel [x] = ([x], [])
  | unravel (x :: y :: xs) =
      let
        val (a, b) = unravel xs
      in
        (x :: a, y :: b)
      end

fun interleave ([], ys) = ys
  | interleave (xs, []) = xs
  | interleave (x :: xs, y :: ys) = x :: y :: interleave (xs, ys)
```

Task 9.1. (Recommended)

For each of `unravel` and `interleave`: Write recurrences for the work and span (state clearly what variable the recurrence is written in terms of). Solve the recurrences to find the smallest big-O class containing the respective work and span functions.

Solution 1.

For `unravel`:

Let $n =$ `length L`, where `L` is the input list.

Recurrence:

$$W_{\texttt{unravel}}(0) = k_0$$
$$W_{\texttt{unravel}}(1) = k_1$$
$$W_{\texttt{unravel}}(n) = k_2 + W_{\texttt{unravel}}(n-2)$$

This creates a tree with

- Work/node: k_2
- Nodes/level: 1
- Number of Levels: $\frac{n}{2}$

Closed form:

$$\begin{aligned} &W_{\texttt{unravel}}(n) \\ &= \sum_{i=0}^{\frac{n}{2}} k_2 \\ &= \frac{n}{2} k_2 \\ &\in O(n) \end{aligned}$$

There is no opportunity for parallelism, so the span is the same.

For `interleave`:

Let $n =$ `length L`, where `L` is the first input list. Let $m =$ `length R`, where `R` is the second input list.

Recurrence:

$$
\begin{aligned}
W_{\texttt{interleave}}(0, m) &= k_0 \\
W_{\texttt{interleave}}(n, 0) &= k_1 \\
W_{\texttt{interleave}}(n, m) &= k_2 + W_{\texttt{interleave}}(n-1, m-1)
\end{aligned}
$$

This creates a tree with

- Work/node: k_2
- Nodes/level: 1
- Number of Levels: $\min(m, n)$

Closed form:

$$
\begin{aligned}
& W_{\texttt{interleave}}(n) \\
&= \sum_{i=0}^{\min(m,n)} k_2 \\
&= (\min(m,n))k_2 \\
&\in O(\min(m,n))
\end{aligned}
$$

There is no opportunity for parallelism, so the span is the same.

Task 9.2. (Recommended)

Prove the following lemma (totality of `unravel`):

Lemma 9.1. For all values `L: int list`, `unravel L` is valuable.

This lemma will be useful for the proof in the next task!

Solution 2.

Proof. via structural induction on `L`.

Base Case 1: `L = []`.

$$\texttt{unravel []} \Longrightarrow \texttt{([], [])} \qquad (\texttt{unravel}\text{ clause 1})$$

`([], [])` is a value.

Base Case 2: `L = [x]` for some value `x: int`.

`unravel [x]` $\Longrightarrow$ `([x], [])` (`unravel` clause 2)

`([x], [])` is a value.

Inductive Case: `L = x::y::xs` for some values `x : int, y : int, xs : int list`.

Induction Hypothesis: `f xs` is valuable. Let `(a, b) : int list * int list` be the value such that `f xs` $\hookrightarrow$ `(a, b)`.

`unravel (x::y::xs)` $\Longrightarrow$ `let val (a, b) = unravel xs in (x::a, y::b) end` (`unravel` clause 3)

$\Longrightarrow$ `(x::a, y::b)` (**IH**)

`(x::a, y::b)` is a value.

We have shown for all `L : int list` that `unravel L` is valuable. Therefore, `unravel` is total. □

Task 9.3. (Recommended)

Fill in the numbered blanks to complete the proof below:

We want to show for all values `L : int list`, `interleave (unravel L)` $\cong$ `L`.

Proof. We proceed with structural induction on `L`.

Base Case 1: `L` = __**(B1.1)**__

`interleave (unravel L)`	(given)
$\cong$`interleave (unravel` __**(B1.1)**__ `)`	(`L` assumption)
$\cong$`interleave` __**(B1.2)**__	(__**(B1.3)**__)
$\cong$ __**(B1.1)**__	(__**(B1.4)**__)
$\cong$`L`	(`L` assumption)

Thus, `interleave (unravel L)` $\cong$ `L` for this case.

Base Case 2: `L` = ___**(B2.1)**___

`interleave (unravel L)`	(given)
$\cong$`interleave (unravel` ___**(B2.1)**___ `)`	(`L` assumption)
$\cong$`interleave` ___**(B2.2)**___	(___**(B2.3)**___)
$\cong$ ___**(B2.1)**___	(___**(B2.4)**___)
$\cong$`L`	(`L` assumption)

Thus, `interleave (unravel L)` $\cong$ `L` for this case.

Inductive Case: `L` = `x :: y :: xs` for some values ___**(I.1)**___ .
Inductive hypothesis: Assume ___**(I.2)**___ .

`interleave (unravel L)`	(given)
$\cong$`interleave (unravel (x :: y :: xs))`	(`L` assumption)
$\cong$`interleave (let val (a, b) = unravel xs in (x :: a, y :: b) end)`	(___**(I.3)**___)
$\cong$`interleave (let val (a, b) = v in (x :: a, y :: b) end)`	(v is a value, ___**(I.4)**___)
$\cong$`interleave (x :: a, y :: b)`	(rewrite, `(a,b) = v`)
$\cong$`x :: y ::` ___**(I.5)**___	(interleave clause 3)
$\cong$`x :: y :: (interleave (unravel xs))`	(`(a,b) = v = unravel xs`)
$\cong$`x :: y ::` ___**(I.6)**___	(IH)
$\cong$`L`	(`L` assumption)

Thus, `interleave (unravel L)` $\cong$ `L` for this case.

Then $\forall$`L : int list`, `interleave (unravel L)` $\cong$ `L`. □

Solution 3.

Note: For this solution, because of the symmetry of the base cases, it is fine to swap the order of the B1 and B2 blocks, as long as everything is consistent within each of the blocks.

Base Case 1
(B1.1.) `[]`
(B1.2.) `([], [])`
(B1.3.) `unravel` clause 1
(B1.4.) `interleave` clause 1

Base Case 2
(B2.1.) `[x]` (for the first blank, quantify: for some `x : int`)
(B2.2.) `([x], [])`
(B2.3.) `unravel` clause 2
(B2.4.) `interleave` clause 2

Inductive Case

(I.1) `x : int, y : int, xs : int list`
(I.2) `interleave (unravel xs) = xs`
(I.3) `unravel` clause 3
(I.4) totality of `unravel`
(I.5) `interleave (a,b)`
(I.6) `xs`

10 Trinary Trees (Trees, Work/Span, Structural Induction)

The 15-150 TAs want to plant a garden, but all they had were binary trees. To spice things up, they decided to add an extra branch!

Consider the following function:

```
datatype tritree = Nub | Branch of tritree * tritree * tritree * int

fun leaves Nub = []
  | leaves (Branch (L, C, R, v)) =
    (case (leaves L, leaves C, leaves R) of
        ([],[],[]) => [v]
    | (resL, resC, resR) => resL @ resC @ resR)
```

Notice that the above code takes in a trinary tree and returns a list of all of the leaves in order from left to right.

Task 10.1. (Recommended)

In `code/tritrees/tritrees.sml`, write a function `accleaves` that satisfies the following specification:

> `accleaves : tritree * int list -> int list`
>
> REQUIRES: true
>
> ENSURES: `accleaves (T, L)` = `(leaves T) @ L`

Constraint: You may not use the `leaves` function in your solution.

Solution 1.

```
fun accleaves (Nub : tritree, L : int list) : int list = L
  | accleaves (Branch (Nub, Nub, Nub, v), L) = v::L
  | accleaves (Branch (A, C, R, v), L) =
      accleaves (A, accleaves (C, accleaves (R, L)))
```

Task 10.2.

Write recurrences for the work and span of the `accleaves` and `leaves` functions as functions of the number of Branches in the tree. For `accleaves`, solve the recurrences to find the smallest big-O class containing the work and span functions. Solving the recurrences for `leaves` is difficult, so just try writing out the recurrences.

Assume the trinary trees are balanced.

Solution 2.

For the following recurrences, let n be the number of `Branch`es in the tree.

Work analysis for `leaves`:

$W_{leaves}(0) = c_0$

$W_{leaves}(1) = c_1$

$W_{leaves}(n) = 3W_{leaves}(n/3) + c_2 n$

- There are $\log_3 n$ levels.
- The work per node at level i is $\frac{c_2 n}{3^i}$.
- There are 3^i nodes at level i.

$$\begin{aligned}
\sum_{i=0}^{\log_3 n} 3^i \frac{c_2 n}{3^i} &= \sum_{i=0}^{\log_3 n} c_2 n \\
&= c_2 n \log_3 n \\
W_{leaves}(n) &\in O(n \log n)
\end{aligned}$$

Span analysis for `leaves`:

$S_{leaves}(0) = c_0$

$S_{leaves}(1) = c_1$

$S_{leaves}(n) = S_{leaves}(n/3) + c_2 n$

- There are $\log_3 n$ levels.
- The span per node at level i is $\frac{c_2 n}{3^i}$.
- There is 1 node per level.

$$\begin{aligned}
\sum_{i=0}^{\log_3 n} \frac{c_2 n}{3^i} &\leq \sum_{i=0}^{\infty} \frac{c_2 n}{3^i} \\
&= \frac{c_2 n}{1 - \frac{1}{3}} \\
&= \frac{3 c_2 n}{2} \\
S_{leaves}(n) &\in O(n)
\end{aligned}$$

Note that this summation is more difficult than something we would ask you to solve. You should, however, be able to set up the summation using what you know about the tree method!

Work analysis for `accleaves`:

$W_{accleaves}(0) = c_0$

$W_{accleaves}(1) = c_1$

$W_{accleaves}(n) = 3W_{accleaves}(n/3) + c_2$

- There are $\log_3 n$ levels.

- The work per node at level i is c_2.
- There are 3^i nodes at level i.

$$\sum_{i=0}^{\log_3 n} 3^i c_2 = c_2 \frac{3n-1}{2}$$
$$W_{accleaves}(n) \in O(n)$$

Span analysis for `accleaves`:

$S_{accleaves}(0) = c_0$

$S_{accleaves}(1) = c_1$

$S_{accleaves}(n) = 3S_{accleaves}(n/3) + c_2$

The analysis is identical to the work for `accleaves`, so

$S_{accleaves}(n) \in O(n)$

Task 10.3. (Recommended)

Prove that your definition of `accleaves` is total!

Solution 3.

Let $P(T) \equiv$ `accleaves (T, L)` $\Longrightarrow$ `L'`, where `L'` is a value of type `int list`. We want to show that for all values `T:tritree` and `L:int list`, P(`T`) holds.

We proceed by structural induction on `T`. Throughout the proof, we let `L:int list` be an arbitrary value.

Base Case 1: `T` = `Nub`

`accleaves (Nub, L)` $\Longrightarrow$ `L` (`accleaves` clause 1)

Since `L` is a value, P(`Nub`).

Base Case 2: `T` = `Branch(Nub, Nub, Nub, v)`, where `v` is an arbitrary `int` value.

`accleaves (Branch(Nub, Nub, Nub, v), L)`
$\Longrightarrow$ `v::L` (`accleaves` clause 1)

Since `L` and `v` are values, `v::L` is also a value and thus, P(`Branch(Nub, Nub, Nub, v)`).

Inductive Hypothesis:
Assume P(`A`), P(`C`), P(`R`) for some values `A:tritree, C:tritree, R:tritree`.

Inductive Case: Let `T = Branch(A,C,R,v)`, where `v:int` is an arbitrary value.

```
   accleaves (Branch(A,C,R,v), L)
⟹ accleaves (A, accleaves (C, accleaves (R, L)))
                                              (accleaves clause 3)
⟹ accleaves (A, accleaves (C, L')), where L':int list is a value.
                                    (IH, since R and L are values)
⟹ accleaves (A, L''), where L'':int list is a value.
                                   (IH, since C and L' are values)
⟹ L''', where L''':int list is a value.   (IH, since A and L'' are values)
```

Thus, for all values `T:tritree` and `L:int list`, P(`T`) holds.

Task 10.4.

You can now use the fact that `accleaves` is total to prove that for all values `T : tritree` and all values `L : int list`, using your definition of `accleaves`,

`(leaves T) @ L` ≅ `accleaves (T, L)`

Solution 4.

Let P(`T`) be the property that `(leaves T) @ L` ≅ `accleaves (T, L)`. We want to show that for all values `T:tritree` and all values `L:int list`, P(`T`) holds.

We proceed by structural induction on `T`. Throughout the proof, we let `L:int list` be an arbitrary value.

Base Case 1: Let `T` = `Nub`. LHS:

```
(leaves T) @ L ≅ (leaves Nub) @ L        (T defn)
               ≅ [] @ L                  (leaves clause 1)
               ≅ L                       (@ defn)
```

RHS:

```
accleaves (T, L) ≅ accleaves (Nub, L)    (T defn)
                 ≅ L                     (accleaves clause 1)
```

Thus, P(`Nub`).

Base Case 2: Let `T = Branch(Nub, Nub, Nub, v)`, where `v:int` is an arbitrary value. LHS:

```
(leaves T) @ L ≅ (leaves Branch(Nub, Nub, Nub, v)) @ L        (T defn)
               ≅ (case ([],[],[]) of ([],[],[]) => [v] ... ) @ L
                                                              (res defn)
               ≅ [v] @ L                                      (case clause 1)
               ≅ v::L                                         (@ defn)
```

RHS:

```
accleaves (T, L) ≅ accleaves (Branch(Nub,Nub,Nub,v), L)       (T defn)
                 ≅ v::L                                       (accleaves clause 2)
```

Thus, P(`Branch(Nub, Nub, Nub, v)`).

Inductive Hypothesis:
Assume P(`A`), P(`C`), P(`R`) for some values `A:tritree, C:tritree, R:tritree`, where `A`, `C`, `R` are not all `Nub`'s.

Inductive Case: Let `T = Branch(A,C,R,v)`, where `v:int` is an arbitrary value.
LHS:

```
accleaves (T, L) ≅ accleaves (Branch(A,C,R,v), L)             (T defn)
                 ≅ accleaves (A, accleaves (C, accleaves (R, L)))
                                                              (accleaves clause 3)
                 ≅ (leaves A) @ accleaves (C, accleaves (R, L))
                                                  (IH, totality of accleaves)
                 ≅ (leaves A) @ (leaves C) @ (leaves R) @ L
                                       (IH twice more, totality of accleaves)
```

RHS:

```
(leaves T) @ L ≅ (leaves Branch(A,C,R,v)) @ L                 (T defn)
               ≅ (case (leaves A, leaves C, leaves R) of ... ) @ L
                                                              (leaves clause 2)
               ≅ (leaves A) @ (leaves C) @ (leaves R) @ L
                  (case clause 2, valuability of leaves A, leaves B, leaves C)
```

The valuability of `leaves A`, `leaves B`, and `leaves C` is justified below:
By the totality of `accleaves`, `accleaves (A, L)` is valuable.
By IH, `accleaves (A, L)` ≅ `(leaves A) @ L`.
Therefore `(leaves A) @ L` is valuable. Thus, `leaves A` must be valuable.

Similarly, `leaves B`, and `leaves C` are also valuable.

Thus, $P(T)$, which concludes the inductive case.

Thus, for all values `T:tritree` and `L:int list`, P(`T`) holds.

After the TAs planted the trees, they decided that they actually looked pretty bad, and decided to have you cut off the extra branch.

Task 10.5.

In `code/tritrees/tritrees.sml`, write a function that satisfies the following specification:

```
trim : tritree -> tree
```

REQUIRES: true

ENSURES: `trim T` returns a binary tree `T'` that contains the left and right children of `T` but removes the center children and all of their descendents.

Constraint: You may not use any helper functions for this task.

Solution 5.

```
fun trim (Nub : tritree) : tree = Empty
  | trim (Branch (L, _, R, v)) = Node (trim L, v, trim R)
```

11 haha cost analysis go wrrk (Work/Span)

Recall the tree datatype:

```
datatype tree = Empty | Node of tree * int * tree
```

Consider the following function, which returns a list of all the even numbers in a tree:

```
fun findEvens Empty = []
  | findEvens (Node (L, x, R)) =
      if x mod 2 = 1
  then (case findEvens L of
      [] => findEvens R
        | (y::ys) => (findEvens L) @ (findEvens R))
  else (case findEvens L of
      [] => x::(findEvens R)
        | (y::ys) => (findEvens L) @ (x::(findEvens R)))
```

Task 11.1.

Write and solve recurrences for the work and span of `findEvens` in terms of n, the number of nodes in the tree.

For the solution to the work recurrence, leave your answer in summation form (i.e. you don't need to give the final big-O notation). You can assume that the tree is balanced.

Solution 1.

Work Analysis:

$W(0) = k_0$

$W(n) = 3W\left(\frac{n}{2}\right) + W_{@}(n) + k_1$

$W(n) = 3W\left(\frac{n}{2}\right) + k_2 n + k_1$

- There are $\log_2 n$ levels since the tree is balanced.
- The work per node at level i is $\frac{n}{2^i}$.
- There are 3^i nodes at level i.

$\sum_{i=0}^{\log_2 n} \frac{3^i}{2^i} n$

For the curious, the big O bound is:

$O\left(n^{\log 3/\log 2}\right)$

Span Analysis:

$S(0) = k_0$

$S(n) = 2S\left(\frac{n}{2}\right) + S_{@}(n) + k_1$

$S(n) = 2S\left(\frac{n}{2}\right) + k_2 n + k_1$

- There are $\log_2 n$ levels.

- The span per node at level i is $\frac{n}{2^i}$.
- There are 2^i nodes at level i.

$$\sum_{i=0}^{\log_2 n} \frac{2^i}{2^i} n = \sum_{i=0}^{\log_2 n} n$$
$$\in O\left(n \log n\right)$$

Some 150 TA realized the `findEvens` function above was actually really inefficient. Since they had too much time on their hands, they decided to rewrite it:

```
fun findEvens Empty = []
  | findEvens Node (L, x, R) =
    let
      val (evenL, evenR) = (findEvens L, findEvens R)
    in
      if x mod 2 = 1
      then evenL @ evenR
      else evenL @ (x::evenR)
    end
```

Task 11.2.

Assuming the tree is balanced, write and solve the work and span recurrences for this new and improved implementation of `findEvens` in terms of n, the number of nodes in the tree.

Do the analysis for both the balanced and unbalanced cases. Write and solve the work and span recurrences for this `findEvens` implementation.

Solution 2.

Balanced Case:

Work Analysis:

$W(0) = k_0$

$W(n) = 2W\left(\frac{n}{2}\right) + W_{@}(n) + k_1$

$W(n) = 2W\left(\frac{n}{2}\right) + k_2 n + k_1$

- There are $\log_2 n$ levels.
- The work per node at level i is $\frac{n}{2^i}$.
- There are 2^i nodes at level i.

$$\sum_{i=0}^{\log_2 n} \frac{2^i}{2^i} n = \sum_{i=0}^{\log_2 n} n$$
$$\in O\left(n \log n\right)$$

Span Analysis:

$S\left(0\right) = k_0$

$S\left(n\right) = S\left(\frac{n}{2}\right) + S_{\texttt{@}}\left(n\right) + k_1$

$S\left(n\right) = S\left(\frac{n}{2}\right) + k_2 n + k_1$

- There are $\log_2 n$ levels.
- The span per node at level i is $\frac{n}{2^i}$.
- There is 1 node per level.

$$\sum_{i=0}^{\log n} \frac{n}{2^i} \in O\left(n\right)$$

Unbalanced Case:

We assume the tree has only left children.

Work Analysis:

$W\left(0\right) = k_0$

$W\left(n\right) = W\left(n-1\right) + W_{\texttt{@}}\left(n\right) + k_1$

$W\left(n\right) = W\left(n-1\right) + k_2 n + k_1$

- There are n levels.
- The work per node at level i is $n - i$.
- There is 1 node per level.

$$\sum_{i=0}^{n} (n-i) = \frac{n(n-1)}{2}$$
$$\in O\left(n^2\right)$$

Span Analysis:

$S(0) = k_0$

$S(n) = S(n-1) + S_{\texttt{@}}(n) + k_1$

$S(n) = S(n-1) + k_2 n + k_1$

The analysis is identical to the work for `findEvens` in the unbalanced case, so the big O bound is

$O(n^2)$

We will now define a new `listTree` datatype, which is a tree that contains an `int list` at each node:

```
datatype listTree = listEmpty | listNode of listTree * int list *
    listTree
```

Consider the following function, which computes the inorder traversal of all subtrees of a tree:

```
fun inord (Empty : tree) : int list = []
  | inord (Node (L, x, R)) = inord L @ (x :: inord R)

fun subInord (T : tree) : listTree =
 (case T of
    Empty => listEmpty
  | Node (L, x, R) = listNode (subInord L, inord T, subInord R))
```

Task 11.3.

Write and solve recurrences for the work and span of `subInord` in terms of the number of nodes n in the tree. You can assume the tree is balanced.

Solution 3.

Omitting the recurrences, the work of `inord` is $O(n \log n)$ and its span is $O(n)$.

Work Analysis:

$W(0) = k_0$

$W(n) = 2W\left(\frac{n}{2}\right) + W_{\texttt{inord}}(n) + k_1$

$W(n) = 2W\left(\frac{n}{2}\right) + k_2 n \log n + k_1$

- There are $\log_2 n$ levels.
- The work per node at level i is $\frac{n}{2^i} \log_2 \frac{n}{2^i}$.
- There are 2^i nodes at level i.

$$
\begin{aligned}
\sum_{i=0}^{\log_2 n} 2^i \left(\frac{n}{2^i} \log_2 \frac{n}{2^i}\right) &= \sum_{i=0}^{\log_2 n} n \log_2 \frac{n}{2^i} \\
&= \sum_{i=0}^{\log_2 n} n \left((\log_2 n) - i\right) \\
&= \sum_{i=0}^{\log_2 n} n \cdot i \qquad (*) \\
&= n \left(\frac{(\log_2 n)^2 + \log_2 n}{2}\right) \\
&\in O(n(\log n)^2)
\end{aligned}
$$

To justify (*), note that as i ranges from 0 to $\log_2 n$, $((\log_2 n) - i)$ ranges from $\log_2 n$ to 0. The step is essentially rearranging the sum.

Span Analysis:

$S(0) = k_0$

$S(n) = \max(S\left(\frac{n}{2}\right), S_{\texttt{inord}}(n)) + k_1$

$S(n) = \max(S\left(\frac{n}{2}\right), k_2 n) + k_1$

First, we find an upper bound on $S(n)$. Let's solve this easier recurrence:

$S(n) \leq S\left(\frac{n}{2}\right) + k_2 n + k_1$

- There are $\log_2 n$ levels.
- The span per node at level i is $\frac{n}{2^i}$.
- There is 1 node per level.

$$\sum_{i=0}^{\log n} \frac{n}{2^i} \in O(n)$$

For all $a, b \in \mathbb{R}$, it's true that $\max(a, b) \geq b$. Thus, the original recurrence tells us $S(n) \geq k_2 n + k_1$. Therefore $O(n)$ is the tightest big-O bound.

Task 11.4.

Based on the big-O bound you found in the previous task, is there a way to improve the time complexity of `subInord`?

If so, modify the implementation as you see fit and solve the work and span recurrences for your version of the function. Otherwise, explain why it can't be improved.

Solution 4.

Here is the revised code. We use a helper function `getList : listTree -> int list` which just extracts the list at the root of a listTree.

```
fun subInord' Empty = listEmpty
  | subInord' (Node (L, x, R)) =
  let
    val (L', R') = (subInord' L, subInord' R)
    fun getList listEmpty = []
      | getList (listNode (_, xs, _)) = xs
  in
    listNode (L', getList L' @ (x :: getList R'), R')
  end
```

Omitting the recurrences, the work and span of `@` is $O(n)$.

Work Analysis:

$W(0) = k_0$

$W(n) = 2W\left(\frac{n}{2}\right) + W_{@}(n) + k_1$

$W(n) = 2W\left(\frac{n}{2}\right) + k_2 n + k_1$

- There are $\log_2 n$ levels.
- The work per node at level i is $\frac{n}{2^i}$.
- There are 2^i nodes at level i.

$$\sum_{i=0}^{\log n} n \in O(n \log n)$$

Span Analysis:

$S(0) = k_0$

$S(n) = S\left(\frac{n}{2}\right) + S_{@}(n) + k_1$

$S(n) = S\left(\frac{n}{2}\right) + k_2 n + k_1$

- There are $\log_2 n$ levels.
- The span per node at level i is $\frac{n}{2^i}$.
- There is 1 node per level.

$$\sum_{i=0}^{\log n} \frac{n}{2^i} \in O(n)$$