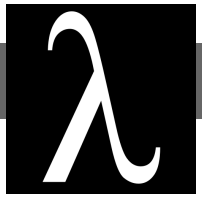

Contents

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.

Task 1.2.

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

Task 1.3.

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

Task 1.4.

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

Task 1.5.

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

Task 1.6.

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

Task 1.7.

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

Task 1.8.

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

Task 1.9.

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

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

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)
```

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`.

Task 1.12.

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

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`.

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`.

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

```
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)
```

1.3 Lists/Datatypes

Task 1.16.

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

Task 1.17.

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

Task 1.18.

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

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

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
```

Task 1.20.

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

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]
```

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))
```

Task 1.23.

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

Task 1.24.

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

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.

Task 1.26.

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

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
```

Task 2.2.

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

Task 2.3. (Recommended)

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

Task 2.4. (Recommended)

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

Task 2.5.

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

Task 2.6. (Recommended)

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

Task 2.7.

```
val v = ()
```

Task 2.8. (Recommended)

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

Task 2.9.

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

Task 2.10.

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

Task 2.11. (Recommended)

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

Task 2.12.

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

Task 2.13. (Recommended)

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

Task 2.14.

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

Task 2.15. (Recommended)

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

Task 2.16. (Recommended)

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

Task 2.17.

```
fun v (b : bool) =
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`

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.

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`

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 = [[]]
```

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: `a`, `b` ≥ 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!

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`

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

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`.

Task 4.2. (Recommended)

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

Task 4.3. (Recommended)

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

Task 4.4. (Recommended)

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

Task 4.5.

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

Task 4.6.

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

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`.

Task 4.8.

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

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 valuable `l1, l2 : int list`, `listsum (l1@l2)` ≅ `listsum l1 + listsum l2`.

Now let's prove the following theorem:

Theorem 5.2. For all values `t : tree`, `treesum t` ≅ `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)`

□

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
```

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`?

Task 6.3.

What is the value that binds to `meat`?

Task 6.4.

What is the value that binds to `cases`?

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`.

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`.

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`,

$$\texttt{rev (L1 @ L2)} \cong \texttt{(rev L2) @ (rev L1)}$$

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

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.

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!

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)
≅interleave (unravel (B2.1) )                              (L assumption)
≅interleave (B2.2)                                         ( (B2.3) )
≅ (B2.1)                                                   ( (B2.4) )
≅L                                                         (L assumption)
```

Thus, `interleave (unravel L)` ≅ `L` for this case.

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

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

Thus, `interleave (unravel L)` ≅ `L` for this case.

Then ∀`L : int list`, `interleave (unravel L)` ≅ `L`. □

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.

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.

Task 10.3. (Recommended)

Prove that your definition of `accleaves` is total!

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`,

$$\texttt{(leaves T) @ L} \cong \texttt{accleaves (T, L)}$$

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.

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.

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.

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.

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.