15-150 Summer 2016
Lab 5

10 February 2016

1 Introduction

The goal for the this lab is to introduce polymophism and defining your own datatypes.

Please take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. You are encouraged to collaborate with your classmates and to ask the TAs for help.

1.1 Getting Started

Update your clone of the `git` repository to get the files for this weeks lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

1.2 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have a purpose and tests.

2 Warmup

Task 2.1 Give the most general (possibly polymorphic) type for the following functions and expressions, or state that the function/expression is ill-typed

1. `let fun fact x = "functions are " ^ x in fact "values " end`

   Solution 2.1 `string`

2. `fun identity x = x`

   Solution 2.1 `'a -> 'a`

3. `fun f x = f x`

   Solution 2.1 `'a -> 'b`

4. `fun f (x,y) = (f(x,y),f(x,y))`

   Solution 2.1 This function is ill-typed.
   To see this assume that the function was not ill-typed. Then it must have type `'a*'b->'c` for some 'c. But looking at the body of the function, we see the body evaluates to 'c*'c which is a contradiction. Thus the function is ill-typed.

Task 2.2 How many different total functions of polymorphic type `'a -> 'a` can there be? (By "different" we mean "not extensionally equivalent".) List all of them.

Solution 2.2 One, the identity function.

3 Options

Recall that SML has a built in datatype `'a option` defined as follows:

```
datatype 'a option = NONE
  | SOME of 'a
```

Intuitively, options let us deal with computation that may not have a result. As such, the `SOME(x)` case represents the successful computation of a value and the `NONE` case represents a failure to compute the value. Note that a failure here is not necessarily a bad thing, it's just a case that we have to handle. Using options lets us express the possibility of failure in the type system. Since we know when an error case is or isn't possible, we can avoid nasty bugs like null pointer dereferencing.

Task 3.1 Define the function

```
findEven: int list -> int option
```

such that `findEven L` returns `SOME x` where x is the first even number in L or `NONE` if L contains no even numbers. Note you may find the built-in SML function `mod` useful here.

Subset Sum Revisited

Recall from homework 3 the function `subset_sum_cert`. For convenience, the source code is reproduced below:

```
fun subset_sum_cert(nil : int list, s : int) : bool * int list = (s=0,nil)
  | subset_sum_cert(x::xs, s) =
      case subset_sum_cert(xs, s - x) of
        (true, l1) => (true, x::l1)
      | (false, _) => subset_sum_cert(xs,s)
```

Observe that when `subset_sum_cert(L,s)` returns false, there is no subset of L that sums to `s`. Still, because of the type definition, we're required to return `(false, l)` where l is some arbitrary list. This is rather silly as we're ignoring the list all together in the false case. We can clean this code up by using option types!

Task 3.2 Define a new function

```
subset_sum_option:int list * int -> int list option
```

such that `subset_sum_option(L,s)` evaluates to `SOME L'` if there exists a subset L' of L whose elements sum to s and `NONE` if no such subset exists.

Task 3.3 Prove the following theorem:

For all `s:int`, `L:int list`, if `subset_sum_option(L,s)` $\cong$ `SOME(L')` then `sum(L')` $\cong$ `s`.

You may assume that `sum` is defined as follows:

```
fun sum [] = 0
  | sum x::xs = x + sum xs
```

Solution 3.3

Base Case: `L` $\cong$ `nil`
To show:
$\forall$`s:int` if `subset_sum_option(nil,s)` $\cong$ `SOME(L')` then `sum(L')` $\cong$ `s`

We proceed by casing on `s`

1. `s` $\cong$ 0 Then by stepping we have:

   ```
          subset_sum_option([],0)
   ==>    SOME []                       (by first clause)
   ```

   Since we have `L'`$\cong$ `[]` and since `sum [] ==> 0` we have shown what we need to show

2. `s` $\ncong$ 0 Then by stepping we have:

   ```
          subset_sum_option([],s)
   ==>    NONE                          (by second clause)
   ```

   Since we have `subset_sum_option([],s)` $\cong$ `NONE` we have nothing more to show

Thus, in the base case, whenever `subset_sum_option(nil,s)`$\cong$ `SOME(L')` then `sum(L')` $\cong$ `s`.

Inductive Step: `L` $\cong$ `x::xs`
Inductive Hypothesis:
$\forall$`s:int` if `subset_sum_option(xs,s)` $\cong$ `SOME(L')` then `sum(L')` $\cong$ `s`

To Show:
$\forall$`s':int` if `subset_sum_option(x::xs,s')` $\cong$ `SOME(LL)` then `sum(LL)` $\cong$ `s'`

We proceed by stepping through `subset_sum_option`

```
     subset_sum_option(x::xs,s')
==>  case subset_sum_option(xs,s'-x) of
         SOME(L') => SOME(x::L')
           | NONE => subset_sum_option(xs, s')
```

We proceed by case analysis

1. `subet_sum_option(xs,s'-x)` $\cong$ `SOME(L')`
   Lets start by considering stepping through `sum`,

   ```
     sum(x::L')
   ⇒x + sum L'
   ≅x + s' - x          By the IH, takings ≅ s'-x, L' ≅ L'
   ≅s'
   ```

   and so we know `subset_sum_option(x::xs,s')` $\cong$ `SOME(LL)` (where `LL == x::L'` here) and `sum(LL)` $\cong$ `s'`
   and so we are done in this case.
2. `subet_sum_option(xs,s'-x)` $\cong$ `NONE`
   Then we have `subset_sum_option(x::xs,s')` $\cong$ `subset_sum_option(xs,s')` (by the second case).
   - If `subset_sum_option(xs,s')` $\cong$ `NONE` then we are done (since we have nothing to prove for the NONE case).
   - Consider the other case where `subset_sum_option(xs,s')` $\cong$ `SOME(LL)`. By the **IH** (taking `s` $\cong$ `s'`, `L'` $\cong$ `LL`) we have `sum(LL) == s'`.

   Thus we have `subset_sum_option(x::xs,s')` $\cong$ `SOME(LL)` and `sum(LL)` $\cong$ `s'` and so we are done.

Therefore, the theorem holds in the inductive case.

The theorem holds by the principle of structural induction for lists.

Have the TAs check your work before proceeding!

$\boxed{\textbf{Solution 3.3}}$

Base Case: `L` $\cong$ `nil`
To show:
$\forall$`s:int` if `subset_sum_option(nil,s)` $\cong$ `SOME(L')` then `sum(L')` $\cong$ `s`

We proceed by casing on `s`

1. `s` $\cong$ 0 Then by stepping we have:

```
     subset_sum_option([],0)
==>  SOME []                        (by first clause)
```

   Since we have `L'`$\cong$ `[]` and since `sum [] ==> 0` we have shown what we need to show

2. `s` $\ncong$ 0 Then by stepping we have:

```
     subset_sum_option([],s)
==>  NONE                           (by second clause)
```

   Since we have `subset_sum_option([],s)` $\cong$ `NONE` we have nothing more to show

Thus, in the base case, whenever `subset_sum_option(nil,s)`$\cong$ `SOME(L')` then `sum(L')` $\cong$ `s`.

Inductive Step: `L` $\cong$ `x::xs`
Inductive Hypothesis:
$\forall$`s:int` if `subset_sum_option(xs,s)` $\cong$ `SOME(L')` then `sum(L')` $\cong$ `s`

To Show:
$\forall$`s':int` if `subset_sum_option(x::xs,s')` $\cong$ `SOME(LL)` then `sum(LL)` $\cong$ `s'`

We proceed by stepping through `subset_sum_option`

```
     subset_sum_option(x::xs,s')
==>  case subset_sum_option(xs,s'-x) of
         SOME(L') => SOME(x::L')
           | NONE => subset_sum_option(xs, s')
```

We proceed by case analysis

1. `subet_sum_option(xs,s'-x)` $\cong$ `SOME(L')`
   Lets start by considering stepping through `sum`,

`sum(x::L')`	
$\Rightarrow$`x + sum L'`	
$\cong$`x + s' - x`	By the IH, taking`s` $\cong$ `s'-x`, `L'` $\cong$ `L'`
$\cong s'$	

   and so we know `subset_sum_option(x::xs,s')` $\cong$ `SOME(LL)` (where `LL == x::L'` here) and `sum(LL)` $\cong$ `s'`

   and so we are done in this case.

2. `subet_sum_option(xs,s'-x)` $\cong$ `NONE`
   Then we have `subset_sum_option(x::xs,s')` $\cong$ `subset_sum_option(xs,s')` (by the second case).

   - If `subset_sum_option(xs,s')` $\cong$ `NONE` then we are done (since we have nothing to prove for the NONE case).
   - Consider the other case where `subset_sum_option(xs,s')` $\cong$ `SOME(LL)`. By the **IH** (taking `s` $\cong$ `s'`, `L'` $\cong$ `LL`) we have `sum(LL) == s'`.

   Thus we have `subset_sum_option(x::xs,s')` $\cong$ `SOME(LL)` and `sum(LL)` $\cong$ `s'` and so we are done.

Therefore, the theorem holds in the inductive case.

The theorem holds by the principle of structural induction for lists.

4 Senpai no Sekai

In many clubs at college, the most senior member of a club is referred to as "senpai". He or she serves as a highly-respected mentor, who guides their juniors and other underlings. In the world of *Senpai Sekai*, there is a "senpai". Senpai is a nice person. Senpai notices many people, and many people desire to be noticed by senpai.

We will represent the set of people that senpai has noticed using shrubs.

```
datatype shrub = Leaf of person option
               | Branch of shrub * shrub
```

Notice in this tree-like representation, all the data is at the leaves.

Initially, senpai has not yet noticed anyone. We represent this state with a tree where all leaves contain the `NONE` option.

```
fun initial (0 : int) : shrub = Leaf (NONE)
  | initial (depth) =
      let
        val subshrub = initial (depth - 1)
      in
        Branch (subshrub, subshrub)
      end
```

Task 4.1 Derive a recurrence relation for $W_{\texttt{initial}}(d)$ and derive a big-O estimate for the work it takes for `initial` to produce a shrub of depth d.

Solution 4.1

$$W_{\texttt{initial}}(0) = k_1$$
$$W_{\texttt{initial}}(d) = k_2 + W_{\texttt{initial}}(d-1)$$

$W_{\texttt{initial}}(d) = k_1 + nk_2$ which is $O(d)$.

Task 4.2 Derive a recurrence relation for $S_{\texttt{initial}}(d)$ and derive a big-O estimate for the span of producing a shrub of depth d.

Solution 4.2

$$S_{\texttt{initial}}(0) = k_1$$
$$S_{\texttt{initial}}(d) = k_2 + S_{\texttt{initial}}(d-1)$$

$S_{\texttt{initial}}(d) = k_1 + nk_2$ which is $O(d)$.

Consider the function,

```
fun senpaiNotices (Leaf NONE : shrub, [] : person list) = (Leaf NONE, [])
  | senpaiNotices (Leaf NONE, x :: xs) = (Leaf (SOME x), xs)
  | senpaiNotices (Branch (L, R), people) =
      let
        val (leftShrub, peopleLeft) = senpaiNotices (L, people)
        val (rightShrub, peopleLeft') = senpaiNotices (R, peopleLeft)
      in
        (Branch (leftShrub, rightShrub), peopleLeft')
      end
```

which takes an empty shrub and a list of people who desire to be noticed by senpai, and returns a shrub filled with people whom senpai has noticed, along with a list of people who still are waiting to be noticed.

Task 4.3 Derive a recurrence relation for $W_{\texttt{senpaiNotices}}(d)$ for the worst case, expressed as a big-O estimate in terms of a d, where d is the depth of the shrub. Assume that senpai is powered by only a single processor, how long will it take for senpai to notice you?

$\boxed{\textbf{Solution 4.3}}$

$$W_{\texttt{senpaiNotices}}(0) = k_1$$
$$W_{\texttt{senpaiNotices}}(d) = k_2 + 2 * W_{\texttt{senpaiNotices}}(d-1)$$

$W_{\texttt{senpaiNotices}}(d) = 2^d k_1 + (2^d - 1)k_2$ which is $O(2^d)$.

Task 4.4 Derive a recurrence relation for $S_{\texttt{senpaiNotices}}(d)$ for the worst case, expressed as a big-O estimate in terms of a d, where d is the depth of the shrub. Assume that senpai is powered by infinite processors, how long will it take for senpai to notice you?

$\boxed{\textbf{Solution 4.4}}$

$$S_{\texttt{senpaiNotices}}(0) = k_1$$
$$S_{\texttt{senpaiNotices}}(d) = k_2 + 2 * S_{\texttt{senpaiNotices}}(d-1)$$

$S_{\texttt{senpaiNotices}}(d) = 2^d k_1 + (2^d - 1)k_2$ which is $O(2^d)$.

Task 4.5 Did senpai perform better when equipped with more processors? How could the parameter types and body of the function `senpaiNotices` be changed so that senpai is able to use his processors to notice you more efficiently?

$\boxed{\textbf{Solution 4.5}}$ If the person list was instead a balanced binary tree, we could split the tree into two balanced subtrees and parallelize the two calls to `senpaiNotices`.

5 Jpop World

Senpai likes to listen to music. Unfortunately, senpai's music collection is disorganized, so he cannot listen to the music he likes. We will help senpai by using higher-order function to organize his music in the hopes that he will notice us.

Anonymous functions are function literals that are not explicitly bound to a name. They provide a useful way to define functions inline and will be used extensively with curried and higher-order functions, both of which will be discussed in lecture tomorrow. Anonymous functions may also be called **lambda expressions**. They have the following syntax:

`(fn (x,y) => x + y)` is an anonymous function that adds two integers.

Consider a function bound to the name `add` that adds two integers:

```
fun add (x,y) = x + y
```

Since functions are values, the same function can also be defined as follows:

```
val add = (fn (x,y) => x + y)
```

To evaluate an anonymous function applied to an argument, you plug the value of the argument in for the variable like always:

$$\texttt{(fn (x,y)} \Rightarrow \texttt{x + y) (2 , 3)} \tag{1}$$
$$\Longrightarrow \texttt{2 + 3} \tag{2}$$
$$\Longrightarrow \texttt{5} \tag{3}$$

It is also important to note that the body of the anonymous function cannot be evaluated until the function is applied to an argument. The function body can be thought of as a piece of text that is not evaluated in any way until the function is called. We'll see later in the semester that this "lazy" way of evaluating functions can be very useful.

We will represent senpai's music collection as a list of songs, where each song has a title, artist, and duration.

```
type title = string
datatype artist = Akb | MomoiroCloverZ | MorningMusume
type duration = int
datatype song = Song of title * artist * duration
type musicCollection = song list
```

Consider a function bound to the name `isMomoclo` that check the artist of a song is Momoiro Clover Z:

```
val isMomoclo = fn Song (_, artist, _) => (artist = MomoiroCloverZ)
```

Task 5.1 Implement the following functions by binding each variable name to an appropriate anonymous function.

1. `val containsKoi: song -> bool` checks if the name of a song contains "koi"
2. `val isShort: song -> bool` checks if the duration of a song is shorter than 4 mins

Consider the following functions that take a `musicCollection` as argument and filter the list such that they retain the elements that satisfy a certain condition.

```
fun filterMomoclo ([] : musicCollection) : musicCollection = []
  | filterMomoclo (x :: xs) =
    if isMomoclo x then x :: (filterMomoclo xs) else filterMomoclo xs
fun filterKoi ([] : musicCollection) : musicCollection = []
  | filterKoi (x :: xs) =
    if containsKoi x then x :: (filterKoi xs) else filterKoi xs
```

Both these functions follow a similar pattern and only differ in the condition on the basis of which they are filtering elements from the `song list`. Would it then be possible to generalize the function such that it takes in an `song list` and a condition as arguments and filters the list based on the condition?

Yes, this is possible as functions are values that can be passed to, and returned from other functions. Functions that accept other functions as arguments or return them as results are called **Higher-Order Functions**. Higher-Order Functions are very powerful and allow functional languages to do amazing things in beautifully short lines of code!!!!

Now that we know about high-order functions, let's implement a higher-order filter function so that senpai can notice us.

Task 5.2 Write a higher-order function filter that generalizes the functions `filterMomoclo` and `filterKoi` above. It will have the following type:

```
filter: ((song -> bool) * musicCollection) -> musicCollection
```

Task 5.3 Test the `filter` function you wrote using the functions you wrote. You can also test it by writing the anonymous function inline within the test case. Try both to get used to the syntax.

Task 5.4 We've talked about polymorphism and higher-order functions. Now you will combine these ideas and write a polymorphic version of the higher-order `filter` you just wrote. It should have the type:

```
filter': (('a -> bool) * 'a list) -> 'a list
```

6 Propositions

SML's datatype mechanism is very useful for representing a variety of different structures. So far, you have mostly seen it used for data structures (trees) and some more informative types (option types), but it can also be used to represent other interesting things, such as a simple syntax for logic.

In logic, a *proposition* may be defined as one of the following:

- True
- False
- $A \wedge B$ ("A and B"), where A and B are propositions
- $A \vee B$ ("A or B"), where A and B are propositions
- $A \implies B$ ("if A, then B"), where A and B are propositions
- $\neg A$ ("not A"), where A is a proposition

Note that this fits exactly with SML's datatype mechanism – a proposition is one of many things, several of which can also have propositions as subterms.

Task 6.1 Define a datatype `prop` that corresponds to logical propositions as defined above.

Task 6.2 Define the function

```
eval_prop: prop -> bool
```

such that `eval_prop` returns true if the proposition is true and false otherwise.

Have the TAs check your work before proceeding!

7 Binary Operators Are Values

Consider a datatype `binop` defined as follows:

```
datatype binop = Add
  | Subtract
  | Multiply
```

We can use this datatype to define a simple form of integer arithmetic where any binary operation is one of addition, subtraction, or multiplication. We can translate expressions in this arithmetic into SML as follows:

```
fun eval_binop (b : binop, x : int, y : int) : int =
    case b of
        Add => x + y
      | Subtract => x - y
      | Multiply => x * y
```

Task 7.1 Using the above `eval_binop` function, define a function

```
foldl_binop : binop * int list -> int
```

such that `foldl_binop (b,L)` evaluates to the combination of all elements of `L` using `b`, starting at the left of `L`. *Hint:* Think carefully about your base case.

Recall the operation tree from the very first lab:

```
      *
    /   \
   +     +
  / \   / \
 1   2 3   *
          / \
         4   5
```

We can use our `binop` datatype to evaluate operation trees. First, consider the following datatype:

```
datatype shrub = Leaf of int
  | Node of (shrub, int, shrub)
```

Task 7.2 Define an analogous datatype `binop_tree` which is either an int or a node with a binop and two more `binop_tree`s.

Task 7.3 Define a function

```
eval_tree : binop_tree -> int
```

such that `eval_tree t` will evaluate the operation tree `t`.

8 Signs Point to No

Suppose we wanted to work with integers, but someone sabotaged our copy of sml and removed all the integer comparison operations, including `Int.compare`.

We can get around this by defining a new datatype `sint`:

```
datatype sint = Zero
  | Pos of int
  | Neg of int
```

Where `Zero` denotes 0, `Pos(x)` denotes a positive number with absolute value `x`, and `Neg(x)` denotes a negative number with absolute value `x`. Note that negative integers and 0 are invalid arguments for the `Pos` and `Neg` constructors. We will examine ways to ensure that invalid values are never created in a few weeks, but for now your functions should simply require that no input is given in this form.

Since built-in integer comparison operations have been sabotaged, you may not use them in the following tasks.

Task 8.1 Define functions

```
sint_plus : sint * sint -> sint
```

and

```
sint_minus : sint * sint -> sint
```

which compute addition and subtraction of `sint` values, respectively. You may find the function `Int.sign` useful. It returns `~1` if the parameter is negative, `1` if it is positive, and `0` if it is 0.

It is possible to create a full set of arithmetic operations for this datatype, but we only need these two.

Task 8.2 Define a function

```
sint_compare : sint * sint -> order
```

such that `sint_compare (X,Y)` returns `LESS` if `X` is less than `Y`, `GREATER` if `X` is greater than `Y`, and `EQUAL` if `X` is equal to `Y`.

9 Onii-san's Challenge

Anshu Onii-san challenges you to solve these problems to gain access to his popcorn and the six exotic spices hidden within.

Task 9.1 Give the most general (possibly polymorphic) type for the following functions and expressions, or state that the function/expression is ill-typed

1.
```
(fn y => (fn (f, x) => f (x))
         ((fn x => y),  (fn (x, y) => x + y) (15, 150)))
```

$\boxed{\textbf{Solution 9.1}}$ `'a -> 'a`

Task 9.2 Solve the following recurrences and give the tight Big-O bound.

1. $W(1) = c_0$
   $W(n) = 3 * W(n/2) + c_1 n^2$

   $\boxed{\textbf{Solution 9.2}}$ $O(n^2)$

2. $W(1) = c_0$
   $W(n) = 4 * W(n/2) + c_1 n^2$

   $\boxed{\textbf{Solution 9.2}}$ $O(n^2 \log n)$

3. $W(1) = c_0$
   $W(n) = 5 * W(n/2) + c_1 n^2$

   $\boxed{\textbf{Solution 9.2}}$ $O(n^{\log_2 5})$

4. $W(1) = c_0$
   $W(n) = \sqrt{n} * W(\sqrt{n}) + c_1 \sqrt{n}$

   $\boxed{\textbf{Solution 9.2}}$ $O(n \log \log n)$

10 Values Are Functions

One day, Professor Erdmann realizes that "functions are values", and then stumbles on a brilliant idea: since functions are values, why not represent values using functions?

One way to represent the natural numbers (starting with zero) using functions is using Church encoding. **Church Numerals** are a representation of the natural numbers using lambda notation. The method was discovered by Alonzo Church, who first encoded data in the untyped lambda calculus this way.

We think of numbers as functions that take in a base case and a successor function f.
The number 0 can be viewed as a function that applies f to the base case 0 times.
The number 1 can be viewed as a function that applies f to the base case 1 times.
The number 2 can be viewed as a function that applies f to the base case 2 times.
And so on...

$$
\begin{aligned}
0 &\triangleq \texttt{fn (base, succ) => base}\\
1 &\triangleq \texttt{fn (base, succ) => succ base}\\
2 &\triangleq \texttt{fn (base, succ) => succ (succ base)}\\
3 &\triangleq \texttt{fn (base, succ) => succ (succ (succ base))}\\
n &\triangleq \texttt{fn (base, succ) => succ}^{n}\texttt{ (base)}
\end{aligned}
$$

Task 10.1 Give the most general (possibly polymorphic) type for

```
val one = fn (base, succ) => succ base
```

Solution 10.1 `'a * ('a -> 'b) -> 'b`

Task 10.2 Take a moment to take in the beauty to exercise the Church Numerals in the REPL, starting with the examples shown below:

```
one (0, fn x => x + 1)
two ([], fn L => 0 :: L)
three (False, fn p => Not p)
```

Amazingly, we can implement all basic arithmetic using Church numerals. Here is an implementation of a successor function for natural numbers:

```
fun addOne n = fn (base, succ) => succ (n (base, succ))
```

Closely study how this function works. It takes in a number n and returns a function that applies the successor function on a base n times, and then 1 more time. All together, it applies the function $n+1$, which is just what we wanted.

Task 10.3 Give the most general (possibly polymorphic) type for `addOne`

$\boxed{\text{Solution 10.3}}$ `('a * ('b -> 'c) -> 'b) -> 'a * ('b -> 'c) -> 'c`

Let's try to write `addition` that takes two numerals and adds them. If we have n, a function that applies an successor function n times, and we have m, a function that applies an successor function m times, think about how we can construct a function that applies an successor function $n + m$ times,

Task 10.4 Write a higher-order function `addition` that takes two numerals and adds them. We have provided test cases for you. Simply uncomment them when you are ready. (You may use `addOne`, but there is an equally short solution without it.)

Task 10.5 Write a higher-order function `multiplication`. Do not use `addOne` or `addition` in your solution. We have provided test cases for you. Simply uncomment them when you are ready.

An early challenge to lambda calculus was the inability to do subtraction in this mathematical system. A clever solution was finally discovered by one of Church's students, Stephen Kleene, while having his teeth worked on at the dentist's.

Task 10.6 Implement the function `subOne`. We have provided test cases for you. Simply uncomment them when you are ready. It is fine to return zero if you ever go negative, since we only care about natural numbers. (Hint Hint: It is recommended you implement tuples as functions first.)

The skeptics of functional programming say functions are not values. Can you truly have a programming language with only functions? Prove them wrong, and show them that all data can be represented with only functions.

Task 10.7 Design a Church-like representation for binary trees in lambda calculus.

$\boxed{\text{Solution 10.7}}$

`tree ::= Empty | Node of tree * tree`

Empty $\triangleq$ `fn (base, succ) => base`

Node $(L, R) \triangleq$ `fn (base, succ) => succ (`L`(base, succ), `R`(base, succ))`

You are truly a functional-programming guru. Senpai has finally noticed you. Have the TAs check you in.