Homework 2 CS558 Gabriel Urbaitis
101668562

2.1 Polymorphic binary trees

a) The MultiTree has two type constructors, MLeaf and MNode.
MLeafs are of type a, MNodes are of type b and have two subtrees that recurse the structure.
We derive show for testing purposes.

```
data MultiTree a b = MLeaf a
                   | MNode b (MultiTree a b) (MultiTree a b) deriving Show
```

Testing:
I created two MultiTrees to test, myTree and lakinTree.

myTree goes to a depth of 3 on the left side and 2 on the right side. It has Strings at the nodes to label their orientation in relation to the other nodes, an “L” is added to the string of a left child, an “R” to the string of a right child. It has Ints at the leafs to emphasize different types at nodes and leafs.

```
myTree :: MultiTree Int String
myTree = MNode "Top" (MNode "L" (MNode "LL" (MLeaf 1) (MLeaf 2)) (MLeaf 3)) (MNode "R"
(MLeaf 4) (MLeaf 5))
```

lakinTree is the example given for mirror in part c). It goes to depth of 1 on the left and 2 on the right side. It has Chars at the leafs and Ints at the nodes.

```
lakinTree :: MultiTree Char Int
lakinTree = (MNode 1 (MLeaf 'A') (MNode 2 (MLeaf 'B') (MLeaf 'C')))
```

Both MultiTrees were tested with a simple print function.

b) For the function declaration, multiFold takes two function arguments, one which operates on the expressions stored in the leafs, (a -> c), the other which operates on the expression in the Nodes and their subtrees, (b -> c -> c -> c). It also takes the MultiTree it operates on, (MultiTree a b), and returns a single expression of type c.

```
multiFold :: (a -> c) -> (b -> c -> c -> c) -> (MultiTree a b) -> c
```

For the function definition, there are two patterns to match, where the MultiTree is a MLeaf and where it is a MNode.

For the leaf case, the function simply takes the expression stored at the MNode and applies the fLeaf function to the expression.

```
multiFold fLeaf fNode (MLeaf x) = fLeaf x
```

For the node case, we apply the fNode function to the expression stored at the node and recursive calls to multiFold on both subtrees. When the subtrees are both MLeafs, they will return expressions of type c from the fLeaf function. Then fNode will take the expression at the node of type b and the expressions of type c from the multiFold recursive calls and return a single expression of type c, which itself may be the return of a recursive call up to a parent MNode, or if we're at the root, it is the final expression returned.

multiFold fLeaf fNode (MNode y t1 t2) = fNode y (multiFold fLeaf fNode t1) (multiFold fLeaf fNode t2)

Testing: As I had trees with different types at the leafs and nodes, I needed different fNode and fLeaf functions to test them with multiFold. For myTree, my fLeaf function multiplied the expression at the Leaf by itself to do something unique to numbers, but as it had to return type c that was compatible with Strings, I applied show to the result to return a string of the result. myTree already had a String expression stored at the node, so no transformation was required to turn it to type c. I just appended the String returned by the left child to the String at the Node and appended the string returned by the right child to that string for my fNode function.

myTree = MNode "Top" (MNode "L" (MNode "LL" (MLeaf 1) (MLeaf 2)) (MLeaf 3)) (MNode "R" (MLeaf 4) (MLeaf 5))

multiFold (\x -> show(x*x)) (\y t1Str t2Str-> y ++ t1Str ++ t2Str) myTree =
"TopLLL149R1625"

multiFold at the first return appended show(3*3)=['9'] to "LL" ++ multiFold MLeaf 1 multiFold MLeaf 2 = "LL"++"1"++"4"++"9", At the next level, "L"++"LL14" ++"9". On The Right Side, "R"++show(4*4)++show(5*5) = "R1625". At the Top Level, "Top"++ "LLL149"++ "R1625" = "TopLLL149R1625".

I realized that myTree was only evaluating (c->-c->c->c) as the nodes already had expressions of type String and that was the return type for the whole multiFold function implementation, so I also tested lakinTree.

lakinTree =
MNode 1 (MLeaf 'A') (MNode 2 (MLeaf 'B') (MLeaf 'C'))

multiFold (\x -> [x]) (\y t1Str t2Str-> (show y) ++ t1Str ++ t2Str) lakinTree =
"1A2BC"

With lakinTree having chars at the leafs and Ints at the Nodes, I decided the unique third return type should be Strings (or lists of chars). So my anonymous function for fLeaf took the Char at the leaf and returned a singleton list of that Char (a String). My anonymous function for fNode took the Int at the node and applied show to it to return a String of the number. Then it appended the Strings returned by the subtrees of the node (already explained in depth for myTree). The Left subtree returned “A”, the right subtree returned “2BC” so these appended to the root of “1” returned a String of “1A2BC”

c) For the function declaration mirror takes a MultiTree and returns a mirrored MultiTree consisting of the same types

mirror :: (MultiTree a b) -> (MultiTree a b)

Because we are using multiFold to define mirror, we have to look at how multifold uses the functions to determine what those functions should be.

When multiFold is applied to the expression at the leaf, it returns the fLeaf function applied to that expression. We are trying to construct a new MultiTree to return, so given the expression at the leaf, we create a new MLeaf of that expression.

When multiFold is applied to the expressions at the node, it returns the fNode function applied to the expression at the node and the values returned by the recursive calls of multiFold on the subTrees. Once again we are constructing a MultiTree, so we apply MNode to the expression at the node, but we swap the position of the subtrees in our call to build the mirrored version of the tree. The recursive calls in MultiFold take care of the different instances where the subtrees are leafs or nodes, so we don’t need to worry about that in mirrors definition. So we take in a MultiTree , and run the function for reconstructing leaves and for reconstructing nodes with mirrored children on that Multi Tree.

mirror mTree = multiFold (\x -> MLeaf x) (\y t1 t2 -> MNode y t2 t1) mTree

Testing: I ran mirror on both myTree and the example given in the assignment, lakinTree.

On myTree, mirror produced a tree with the top node labeled the same as the input tree, then on the left child “R” instead of where it was the right child on the input tree. The Children of “R” were leaves 5 and 4, and now the children should, going from left to right, descend instead of ascend. They do, as that portion of the tree is finished, the right node is now “L”, its left child is MLeaf 3, its right child is “LL” and “LL” ’s children are now 2 on the left and 1 on the right.

mirror myTree =
MNode "Top" (MNode "R" (MLeaf 5) (MLeaf 4)) (MNode "L" (MLeaf 3) (MNode "LL" (MLeaf 2) (MLeaf 1)))

Mirro lakinTree also produced a mirrored tree, identical to the expected output in the assignment.

mirror lakinTree =
MNode 1 (MNode 2 (MLeaf 'C') (MLeaf 'B')) (MLeaf 'A')

2.2 Proving program properties

```
product :: [Int] -> Int
product [] = 1
product (x:xs) = x * (product xs)

foldr:: (a->b->b) ->b -> [a] ->b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

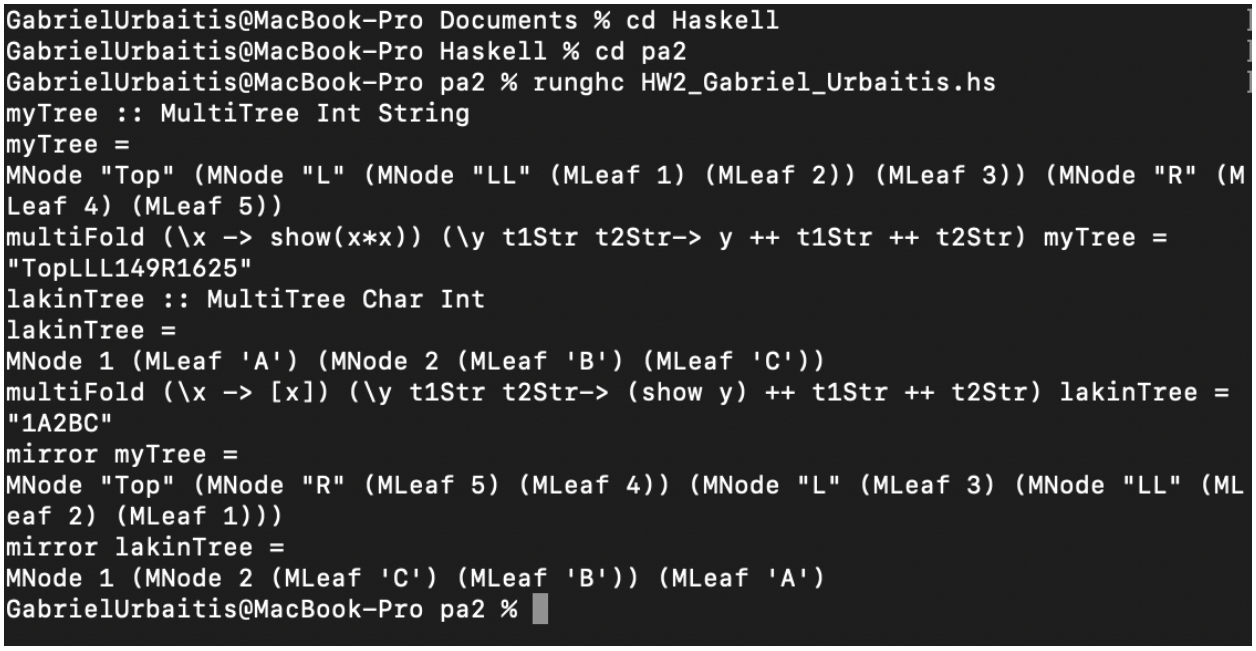
Provide a formal proof that
product ns = foldr (\x -> \acc -> x * acc) 1 ns for all lists ns :: [Int], by induction on the structure of lists.

Base case P([]):
Must show P([]) is product [] = foldr (\x -> \acc -> x * acc) 1 []
LHS = product [] = 1 (From the definition of product)
RHS = foldr (\x -> \acc -> x * acc) 1 [] = 1 (From the definition of foldr)
Hence LHS = 1 = RHS so done.

Inductive Case: ∀x::a. ∀ns::[a]. P(ns) => P (x:ns)
Must show P(ns) -> P (x:ns)
Assume Inductive Hypothesis P(ns) ie product ns = foldr (\x -> \acc -> x * acc) 1 ns
Use to show P(x:ns) ie product (x:ns) = foldr (\x -> \acc -> x * acc) 1 (x:ns)

```
RHS = foldr (\x -> \acc -> x * acc) 1 (x:ns)
    = (\x -> \acc -> x * acc) x (foldr (\x -> \acc -> x * acc) 1 ns)    (From the definition of foldr)
    = (\x -> \acc -> x * acc) x (product ns)     (Using the Inductive Hypothesis)
    = x * (product ns)          (Simplify; Apply the anonymous function)
    = product (x:ns)            (From the definition of product)
    = LHS
So RHS = LHS, so done!
```

Terminal Screenshot:



```
GabrielUrbaitis@MacBook-Pro Documents % cd Haskell
GabrielUrbaitis@MacBook-Pro Haskell % cd pa2
GabrielUrbaitis@MacBook-Pro pa2 % runghc HW2_Gabriel_Urbaitis.hs
myTree :: MultiTree Int String
myTree =
MNode "Top" (MNode "L" (MNode "LL" (MLeaf 1) (MLeaf 2)) (MLeaf 3)) (MNode "R" (M
Leaf 4) (MLeaf 5))
multiFold (\x -> show(x*x)) (\y t1Str t2Str-> y ++ t1Str ++ t2Str) myTree =
"TopLLL149R1625"
lakinTree :: MultiTree Char Int
lakinTree =
MNode 1 (MLeaf 'A') (MNode 2 (MLeaf 'B') (MLeaf 'C'))
multiFold (\x -> [x]) (\y t1Str t2Str-> (show y) ++ t1Str ++ t2Str) lakinTree =
"1A2BC"
mirror myTree =
MNode "Top" (MNode "R" (MLeaf 5) (MLeaf 4)) (MNode "L" (MLeaf 3) (MNode "LL" (ML
eaf 2) (MLeaf 1)))
mirror lakinTree =
MNode 1 (MNode 2 (MLeaf 'C') (MLeaf 'B')) (MLeaf 'A')
GabrielUrbaitis@MacBook-Pro pa2 %
```

3.1 a) if(if true then false else true) then (pred false) else 0

$$\dfrac{\dfrac{\dfrac{}{\text{true}:\text{Bool}}\,\text{(T-True)}\quad \dfrac{}{\text{false}:\text{Bool}}\,\text{(T-False)}\quad \dfrac{}{\text{true}:\text{Bool}}\,\text{(T-True)}}{\text{if true then false else true}:\text{Bool}}\,\text{(T-IF)}\quad \dfrac{\text{false}:\text{Bool}}{\text{pred false}:}\,\text{(T-PRED)}\quad \dfrac{}{0:\text{Nat}}\,\text{(T-Zero)}}{\text{if(if true then false else true) then (pred false) else 0}}\,\text{(T-IF)}$$

Fail Here: (T-PRED) requires subterm to have type Nat, but here it is bool. Thus, term is <u>not</u> well-typed.

b) iszero(if(iszero(pred(pred 0))) then (succ(pred 0)) else 0)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{0:\text{Nat}}{\text{pred}\,0:\text{Nat}}\,\text{(T-PRED)}}{\text{pred(pred}\,0):\text{Nat}}\,\text{(T-PRED)}}{\text{iszero(pred(pred}\,0)):\text{Bool}}\,\text{(T-ISZERO)}\quad \dfrac{\dfrac{\dfrac{}{0:\text{Nat}}\,\text{(T-ZERO)}}{\text{pred}\,0:\text{Nat}}\,\text{(T-PRED)}}{\text{succ(pred}\,0):\text{Nat}}\,\text{(T-SUCC)}\quad \dfrac{}{0:\text{Nat}}\,\text{(T-ZERO)}}{\text{if(iszero(pred(pred}\,0))) \text{ then (succ(pred}\,0)) \text{ else }0:\text{Nat}}\,\text{(T-IF)}}{\text{iszero(if(iszero(pred(pred}\,0))) \text{ then (succ(pred}\,0)) \text{ else }0):\text{Bool}}\,\text{(T-ISZERO)}$$

well typed, the term is of type Bool

3.2 a) if(if true then false else true) then (pred false) else 0

Step 1: Find t' such that $t \to t'$

$$\cfrac{\cfrac{}{\overset{t_2}{\text{if true then false else true}} \longrightarrow \text{false}}\,\text{(E-IFTRUE)}}{\text{if(if true then false else true) then (pred false) else 0} \longrightarrow \text{if(false) then (pred false) else 0}}\,\text{(E-IF)}$$

Step 2: find t'' s.t. $t' \to t''$

$$\cfrac{}{\text{if false then (pred false) else 0} \leftarrow t_3 \longrightarrow 0}\,\text{(E-IFFALSE)}$$

reduced to a value! (0)

b) iszero(if (iszero(pred (pred 0))) then (succ (pred 0)) else 0)

Step 1 Find t' s.t. $t \to t'$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\text{pred 0} \to 0}\,\text{(E-PREDZERO)}}{\text{pred(pred 0)} \to \text{pred(0)}}\,\text{(E-PRED)}}{\text{iszero(pred(pred 0))} \to \text{iszero(pred 0)}}\,\text{(E-ISZERO)}}{\text{if(iszero(pred(pred 0))) then (succ(pred 0)) else 0} \to \text{if(iszero(pred 0)) then (succ(pred 0)) else 0}}\,\text{(E-IF)}}{\text{iszero(if(iszero(pred(pred 0))) then (succ(pred 0)) else 0)} \to \text{iszero(if(iszero(pred 0)) then (succ(pred 0)) else 0)}}\,\text{(E-ISZERO)}$$

Step 2 find t'' s.t. $t' \to t''$

$$\cfrac{\cfrac{\cfrac{\text{pred 0} \longrightarrow 0}{\text{iszero(pred 0)} \to \text{iszero 0}}\,\text{(E-ISZERO)}}{\text{if(iszero(pred 0)) then (succ(pred 0)) else 0} \longrightarrow \text{if(iszero 0) then (succ(pred 0)) else 0}}\,\text{(E-IF)}}{\text{iszero(if(iszero(pred 0)) then (succ(pred 0)) else 0)} \longrightarrow \text{iszero(if(iszero 0) then (succ(pred 0)) else 0)}}\,\text{(E-ISZERO)}$$

Step 3: find t''' s.t. $t'' \rightarrow t'''$

$$\dfrac{\dfrac{\dfrac{}{\text{is zero } 0 \longrightarrow \text{true}}\;(\text{E-ISZEROZERO})}{\begin{matrix}\text{if(is zero 0) then} \\ \text{(succ(pred 0)) else 0}\end{matrix} \longrightarrow \begin{matrix}\text{if true then} \\ \text{(succ(pred 0)) else 0}\end{matrix}}\;(\text{E-IF})}{\begin{matrix}\text{is zero(if(is zero 0)} \\ \text{then(succ(pred 0)) else 0)}\end{matrix} \longrightarrow \begin{matrix}\text{is zero (if true then} \\ \text{(succ(pred 0)) else 0)}\end{matrix}}\;(\text{E-ISZERO})$$

Step 4: find t'''' s.t. $t''' \rightarrow t''''$

$$\dfrac{\dfrac{}{\begin{matrix}\text{if true then} \\ \text{(succ(pred 0)) else 0}\end{matrix} \longrightarrow \text{(succ(pred 0))}}\;(\text{E-IFTRUE})}{\begin{matrix}\text{is zero(if true then} \\ \text{(succ(pred 0)) else 0)}\end{matrix} \longrightarrow \text{is zero (succ(pred 0))}}\;(\text{E-ISZERO})$$

Step 5 find t''''' s.t. $t'''' \rightarrow t'''''$

$$\dfrac{\dfrac{\dfrac{}{\text{pred } 0 \longrightarrow 0}\;(\text{E-PREDZERO})}{\text{succ (pred 0)} \longrightarrow \text{succ 0}}\;(\text{E-SUCC})}{\text{is zero (succ(pred 0))} \longrightarrow \text{is zero (succ 0)}}\;(\text{E-ISZERO})$$

Step 6. Find t'''''' s.t. $t''''' \rightarrow t''''''$

$$\dfrac{}{\text{is zero (succ 0)} \longrightarrow \text{false}}\;(\text{E-ISZEROSUCC})$$

reduced to a value! (false)

4.1 a) $(\lambda x.(\lambda y.x))((\lambda z.(\lambda q.(zq)))(\lambda x.x))$

Step 1:

$$\dfrac{\dfrac{}{(\lambda z.(\lambda q.(zq)))(\lambda x.x) \longrightarrow (\lambda q.((\lambda x.x)q))}\text{ E-APPABS}}{\underbrace{(\lambda x.(\lambda y.x))}_{v_1}\underbrace{((\lambda z.(\lambda q.(zq)))(\lambda x.x))}_{t_2} \longrightarrow (\lambda x.(\lambda y.x))(\lambda q.((\lambda x.x)q))}\text{ E APP-2}$$

Step 2:

$$\dfrac{}{(\lambda x(\lambda y.x))(\lambda q.((\lambda x.x)q)) \longrightarrow \underbrace{(\lambda y.\underbrace{(\lambda q.((\lambda x.x)q))}_{t})}_{v = \lambda y.t}}\text{ E-APP ABS}$$

So we have a value! (t cannot be further reduced)

b) $((\lambda f.(\lambda x.(f(fx))))(\lambda y.y))(\lambda z.(zz))$

Step 1:

$$\dfrac{\dfrac{}{(\lambda f.(\lambda x.(f(fx))))(\lambda y.y) \longrightarrow (\lambda x((\lambda y.y)((\lambda y.y)x)))}\text{ E-APPABS}}{\underbrace{((\lambda f.(\lambda x.(f(fx))))(\lambda y.y))}_{t_1}\underbrace{(\lambda z.(zz))}_{t_2} \rightarrow (\lambda x((\lambda y.y)((\lambda y.y)x)))(\lambda z.(zz))}\text{ E-APP1}$$

Step 2:

$$\dfrac{}{(\lambda x((\lambda y.y)((\lambda y.y)x)))(\lambda z.(zz)) \rightarrow ((\lambda y.y)((\lambda y.y)(\lambda z.(zz))))}\text{ E-APPABS}$$

Step 3:

$$\dfrac{}{(\lambda y.y)((\lambda y.y)(\lambda z.(zz))) \rightarrow ((\lambda y.y)(\lambda z.(zz)))}\text{ E-APPABS}$$

Step 4:

$$\dfrac{}{(\lambda y.y)(\lambda z.(zz)) \longrightarrow \underbrace{(\lambda z.\underbrace{(zz)}_{t})}_{v = \lambda z.t}}\text{ E-APPABS}$$

So we have a value! (t cannot be further reduced)

4.2 $nor = \lambda b_1 . \lambda b_2 . b_1 \; fls (b_2 \; fls \; tru)$

b_1 is our testing variable, if it is tru, then we automatically fail, so the first expression (the tru path) must be fls. . . If b_1 is fls , our answer depends on b_2. In that case, if b_2 is tru, we also fail, so the first expression in the subexpression must be fls. If b_2 is fls, then our nor is satisfied, and taking the fls path, we evaluate to tru.

Mid-term examination #1 — given Tuesday 4th October

General instructions

Closed-book, closed-notes, closed-computer, in-class exam.

Time allowed: 75 minutes.

Total points available: 150 pts.

Answer in the spaces provided.

Your name (print):

I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.

Please sign and date:

1.1 Evaluation in Haskell (20 pts)

For each of the following Haskell expressions, indicate whether evaluation of the expression either:

- reduces to a value (in which case, you must **identify that value**),
- fails to terminate, or
- raises a runtime error.

In each case, you must **provide an explanation for your answer.**

1. (5 pts) `(\m -> \n -> n ‘div‘ m) 0`

2. (5 pts) `(\m -> \n -> n ‘div‘ m) 3 9`

3. (5 pts) `(\m -> \n -> n ‘div‘ m) 0 9`

4. (5 pts) `map (3*) (take 5 [1, 2..])`

1.2 Tree operations in Haskell (50 pts)

Here is the definition of a Haskell type of generalized binary trees with exactly two sub-trees per `Node`, with data values of type `a` stored in the `Nodes` and data values of type `b` stored in the `Leafs`:

```
data GenTree a b = Leaf b
                 | Node a (GenTree a b) (GenTree a b) deriving Show
```

where the first and second arguments to `Node` represent the "left" and "right" sub-trees of that node, respectively.

1. (10 pts) Write a Haskell function

   ```
   leafMax :: Ord b => GenTree a b -> b
   ```

   that returns the maximum value stored in any `Leaf` within the tree.

   For example, let `t :: GenTree Int Char` be the Haskell value

   ```
   Node 7 (Leaf 'J') (Node 2 (Leaf 'Z') (Leaf 'C')).
   ```

   Then, the expression `leafMax t` should evaluate to `'Z'`.

2. (5 pts) Briefly explain the meaning of the `Ord b` constraint on the type in the previous question and why it is necessary there.

3. (10 pts) Write a Haskell function

```
nodeMax :: Ord a => GenTree a b -> Maybe a
```

that returns the maximum value stored in any `Node` within the tree.
For example, with `t` defined as above, the expression `nodeMax t` should evaluate to `Just 7`.

4. (5 pts) Briefly explain why the use of a `Maybe` type for the output is necessary in your answer to the previous question.

5. (15 pts) Write a Haskell function

```
depthFirst :: (GenTree a b) -> [Either a b]
```

that produces a list containing the values from the `Leaf`s and `Node`s in a depth-first fashion, recording the value stored in each `Node` before visiting the left and right sub-trees of that `Node` in that order.

For example, with `t` defined as above, the expression `depthFirst t` should evaluate to `[Left 7, Right ’J’, Left 2, Right ’Z’, Right ’C’]`.

6. (5 pts) Briefly explain why the use of an `Either` type in the output is necessary in your answer to the previous question.

1.3 List operations in Haskell (40 pts)

1. (20 pts) Write a Haskell function

   ```
   takeUntil :: (a -> Bool) -> [a] -> [a]
   ```

   that takes a predicate `p` and a list as arguments and returns a list containing all elements of the input list, in the same order, up until the first element `x` of the input list for which `p x` evaluates to `True`. That element `x` should **not** be included in the resulting list. If `p x` **never** evaluates to `True` then **all** of the elements of the input list should be included in the output list.

   For example, `takeUntil (\n -> n ‘mod‘ 2 == 0) [1,3,5,7,2,4,6,7]` should return the list `[1,3,5,7]`.

   For full credit, make minimal use of any related Haskell library functions in your answer.

2. (20 pts) Name and briefly explain the feature of Haskell evaluation means that Haskell programs can manipulate infinite lists without necessarily looping forever.

   Illustrate your answer using the expression: `takeUntil (\n -> n > 11) [2, 4..]`

1.4 Proving properties of Haskell programs (40 pts)

The `product` function in Haskell can be defined as follows:

```
product :: [Int] -> Int
product [] = 1
product (x:xs) = x * product xs
```

Prove, by induction on lists, that

```
foldr (*) 1 xs = product xs
```

for all lists `xs :: [Int]`.

END OF EXAM

Mid-term examination #2 — given Thursday 17th November

General instructions

Closed-book, closed-notes, closed-computer, in-class exam.

Time allowed: 75 minutes.

Total points available: 150 pts.

Answer in the spaces provided.

Your name (print):

I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.

Please sign and date:

2.1 Typing judgments (40 pts)

1. (20 pts) Using the typing rules for the typed calculus of numbers and booleans, as defined in Appendix A, identify the type T such that

$$\mathsf{pred}\,\big(\mathsf{if}\ \mathsf{iszero}\,(\mathsf{succ}\,0)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{succ}\,0\big)\ :\ T$$

   holds, if such a type exists, by **constructing a full typing derivation for this judgment.**

   If no such type exists, **construct a partial derivation and explain where and why the typing derivation fails.**

2. (20 pts) Using the typing rules for the typed calculus of numbers and booleans, as defined in Appendix A, identify the type T such that

$$\mathsf{iszero}\,\big(\mathsf{if}\ \mathsf{iszero}\,(\mathsf{pred}\ 0)\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{succ}\ 0\big)\ :\ T$$

holds, if such a type exists, by **constructing a full typing derivation for this judgment.**

If no such type exists, **construct a partial derivation and explain where and why the typing derivation fails.**

2.2 Evaluation judgments (40 pts)

1. (20 pts) Using the reduction rules for the typed calculus of numbers and booleans, as defined in Appendix A, **construct a full derivation for *every* step of evaluation** of the term

$$\mathsf{pred}\ \big(\mathsf{if}\ \mathsf{iszero}\ (\mathsf{succ}\ 0)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{succ}\ 0\big).$$

   When you have reduced the term to one that cannot be reduced further, **identify whether that term is a value or whether reduction is "stuck".**

   For this question, you should attempt to reduce the term **regardless** of whether it is well-typed or not.

2. (20 pts) Using the reduction rules for the typed calculus of numbers and booleans, as defined in Appendix A, **construct a full derivation for *every* step of evaluation** of the term

$$\mathsf{iszero}\ \big(\mathsf{if}\ \mathsf{iszero}\ (\mathsf{pred}\ 0)\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{succ}\ 0\big).$$

When you have reduced the term to one that cannot be reduced further, **identify whether that term is a value or whether reduction is "stuck".**

For this question, you should attempt to reduce the term **regardless** of whether it is well-typed or not.

2.3 Implementing arithmetic in the untyped lambda-calculus (30 pts)

1. (5 pts) Write down definitions for the untyped lambda-terms corresponding to the Church numerals c_0, c_1, c_2, c_3, and c_4, which represent the natural numbers 0, 1, 2, 3, and 4, respectively.

2. (10 pts) The untyped lambda-term succ that implements the successor function on Church numerals can be defined as follows:

$$\mathsf{succ} = \lambda n.\, \lambda s.\, \lambda z.\, s\ (n\ s\ z)$$

Using this definition of succ, or otherwise, define an untyped lambda-term plus that takes two Church numerals as arguments and returns the Church numeral corresponding to their sum, i.e., $(\mathsf{plus}\ \mathsf{c}_i)\ \mathsf{c}_j$ should evaluate to c_{i+j}.

3. (15 pts) Using your definition of plus from part 2, or otherwise, define an untyped lambda-term times that takes two Church numerals as arguments and returns the Church numeral corresponding to their product, i.e., $(\mathsf{times}\ \mathsf{c}_i)\ \mathsf{c}_j$ should evaluate to c_{i*j}.

2.4 Concepts in syntax and semantics of programming languages (40 pts)

Briefly explain the following concepts in the syntax and semantics of programming languages. **Illustrate each answer with an explanation of a simple example.**

1. (10 pts) Syntax-directed schematic inference rules.

2. (10 pts) Full beta-reduction of untyped lambda-terms.

3. (20 pts) Type safety.

END OF EXAM

A Reference: the typed calculus of numbers and booleans

Syntax

Terms, t	::=	true	*constant true*
	|	false	*constant false*
	|	if t then t else t	*conditional*
	|	0	*constant zero*
	|	succ t	*successor*
	|	pred t	*predecessor*
	|	iszero t	*zero test*
Values, v	::=	true	*true value*
	|	false	*false value*
	|	nv	*numeric value*
Numeric values, nv	::=	0	*zero value*
	|	succ nv	*successor value*
Types, T	::=	Bool	*type of booleans*
	|	Nat	*type of natural numbers*

Evaluation Rules

$$\frac{}{\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow t_2}\ \text{(E-IfTrue)}$$

$$\frac{}{\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow t_3}\ \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3}\ \text{(E-If)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{succ}\ t_1 \longrightarrow \mathsf{succ}\ t_1'}\ \text{(E-Succ)}$$

$$\frac{}{\mathsf{pred}\,0 \longrightarrow 0}\ (\text{E-PredZero})$$

$$\frac{}{\mathsf{pred}\,(\mathsf{succ}\,nv_1) \longrightarrow nv_1}\ (\text{E-PredSucc})$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{pred}\,t_1 \longrightarrow \mathsf{pred}\,t_1'}\ (\text{E-Pred})$$

$$\frac{}{\mathsf{iszero}\,0 \longrightarrow \mathsf{true}}\ (\text{E-IsZeroZero})$$

$$\frac{}{\mathsf{iszero}\,(\mathsf{succ}\,nv_1) \longrightarrow \mathsf{false}}\ (\text{E-IsZeroSucc})$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{iszero}\,t_1 \longrightarrow \mathsf{iszero}\,t_1'}\ (\text{E-IsZero})$$

Typing Rules

$$\frac{}{\mathsf{true} : \mathsf{Bool}}\ (\text{T-True})$$

$$\frac{}{\mathsf{false} : \mathsf{Bool}}\ (\text{T-False})$$

$$\frac{t_1 : \mathsf{Bool} \qquad t_2 : T \qquad t_3 : T}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}\ (\text{T-If})$$

$$\frac{}{0 : \mathsf{Nat}}\ (\text{T-Zero})$$

$$\frac{t : \mathsf{Nat}}{\mathsf{succ}\, t : \mathsf{Nat}}\ (\text{T-Succ})$$

$$\frac{t : \mathsf{Nat}}{\mathsf{pred}\, t : \mathsf{Nat}}\ (\text{T-Pred})$$

$$\frac{t : \mathsf{Nat}}{\mathsf{iszero}\, t : \mathsf{Bool}}\ (\text{T-IsZero})$$

Final examination — given Tuesday 13th December

General instructions

Closed-book, closed-notes, closed-computer, in-class exam.

Time allowed: 120 minutes.

Total points available: 200 pts.

Answer in the spaces provided.

Your name (print):

I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.

Please sign and date:

3.1 Concepts in Haskell and functional programming (30 pts)

Explain the following concepts, **illustrating each answer with a simple example.**

1. (15 pts) partial application of curried functions in Haskell.

2. (15 pts) laziness in Haskell and the manipulation of infinite lists.

3.2 Haskell—proofs on lists (20 pts)

The `product` function in Haskell can be defined as follows:

```
product :: [Int] -> Int
product [] = 1
product (x:xs) = x * product xs
```

Prove, by induction on lists, that

```
foldr (*) 1 xs = product xs
```

for all lists `xs :: [Int]`.

You must **clearly state** your inductive hypothesis in the inductive case of your proof.

3.3 Typing and evaluation in the simply-typed lambda-calculus (40 pts)

1. (15 pts) Using the typing rules for the simply-typed lambda-calculus with numbers and booleans (presented in Appendix A), **attempt to construct a typing derivation for the following term**, starting in the empty typing environment:

$$\big(\mathsf{if}\ \mathsf{iszero}\ (\mathsf{succ}\ 0)\ \mathsf{then}\ (\lambda x\!:\!\mathsf{Nat}.\ \mathsf{iszero}\ (\mathsf{succ}\ x))\ \mathsf{else}\ (\lambda n\!:\!\mathsf{Nat}.\ \mathsf{iszero}\ (\mathsf{pred}\ n))\big)\ 0$$

   If your derivation is successful, clearly state this and identify the resulting type.

   If your derivation fails, construct as much of the derivation as possible and explain where and why the derivation fails to assign a type.

2. (25 pts) Using the call-by-value evaluation rules for the simply-typed lambda-calculus with numbers and booleans (presented in Appendix A), **construct a full derivation for *each* step of reduction of the following term**:

$$\big(\mathsf{if}\ \mathsf{iszero}\ (\mathsf{succ}\ 0)\ \mathsf{then}\ (\lambda x\!:\!\mathsf{Nat}.\ \mathsf{iszero}\ (\mathsf{succ}\ x))\ \mathsf{else}\ (\lambda n\!:\!\mathsf{Nat}.\ \mathsf{iszero}\ (\mathsf{pred}\ n))\big)\ 0$$

Identify when no more reduction steps can be taken and whether the resulting term is a **value** or whether the term is **stuck** at a non-value term.

You should attempt to reduce this term **regardless** of whether you found it to be well-typed or not in part 1.

3.4 Concepts in lambda-calculi (30 pts)

Explain the following concepts, **illustrating each answer with a simple example.**

1. (15 pts) Church numerals and arithmetic in the untyped lambda-calculus.

2. (15 pts) typing environments for assigning types in the simply typed lambda-calculus.

3.5 Representing booleans in the untyped lambda-calculus (40 pts)

1. (10 pts) Following the convention introduced in class, define closed untyped lambda-terms tru and fls that represent the boolean constants true and false, respectively.

2. (15 pts) Define untyped lambda-terms *pair*, *fst*, and *snd* that implement the operations of pair creation, first element projection, and second element projection, respectively, such that the following hold:

$$fst\ (pair\ f\ s) \longrightarrow_{\beta}^{*} f \qquad\qquad snd\ (pair\ f\ s) \longrightarrow_{\beta}^{*} s.$$

You must define any helper functions that you use in your answer.

3. (15 pts) Explain the workings of your *pair*, *fst*, and *snd* terms, as defined in answer to the previous question.

3.6 General recursion in the simply typed lambda-calculus (40 pts)

1. (25 pts) With reference to the evaluation rules from Appendix A and Appendix B, **construct full derivations for the first two steps of evaluation** of the following term in the simply typed lambda-calculus extended with general recursion:

$$\big(\mathsf{fix}\,(\lambda f\!:\!\mathsf{Nat} \to \mathsf{Nat}.\, \lambda n\!:\!\mathsf{Nat}.\, f\ n)\big)\ \mathsf{0}$$

2. (15 pts) **With reference to your answer to the previous question,** summarize and explain the behavior of the recursive function $\mathsf{fix}\,(\lambda f\!:\!\mathsf{Nat} \to \mathsf{Nat}.\, \lambda n\!:\!\mathsf{Nat}.\, f\ n)$.

END OF EXAM

This page intentionally left (almost) blank

A Reference: simply typed lambda-calculus with booleans and numbers

Syntax

Terms, t	$::=$	x	*variable*
	$\|$	$\lambda x\!:\!T.\,t$	*abstraction*
	$\|$	$t\ t$	*application*
	$\|$	true	*constant true*
	$\|$	false	*constant false*
	$\|$	if t then t else t	*conditional*
	$\|$	0	*constant zero*
	$\|$	succ t	*successor*
	$\|$	pred t	*predecessor*
	$\|$	iszero t	*zero test*
Values, v	$::=$	$\lambda x\!:\!T.\,t$	*abstraction value*
	$\|$	true	*true value*
	$\|$	false	*false value*
	$\|$	nv	*numeric value*
Numeric values, nv	$::=$	0	*zero value*
	$\|$	succ nv	*successor value*
Types, T	$::=$	$T \to T$	*type of functions*
	$\|$	Bool	*type of booleans*
	$\|$	Nat	*type of natural numbers*
Typing contexts, Γ	$::=$	$\varnothing$	*empty context*
	$\|$	$\Gamma, x : T$	*variable type assumption*

Capture-Avoiding Substitution

$$\begin{array}{rcl}
[x \mapsto t]x & = & t \\
[x \mapsto t]y & = & y \qquad \text{if } x \neq y \\
[x \mapsto t](t_1\ t_2) & = & ([x \mapsto t]t_1)\ ([x \mapsto t]t_2) \\
[x \mapsto t](\lambda x{:}T.\, t') & = & \lambda x{:}T.\, t' \\
[x \mapsto t](\lambda y{:}T.\, t') & = & \lambda y{:}T.\, ([x \mapsto t]t') \qquad \text{if } x \neq y \\
[x \mapsto t]\mathsf{true} & = & \mathsf{true} \\
[x \mapsto t]\mathsf{false} & = & \mathsf{false} \\
[x \mapsto t](\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) & = & \mathsf{if}\ ([x \mapsto t]t_1)\ \mathsf{then}\ ([x \mapsto t]t_2)\ \mathsf{else}\ ([x \mapsto t]t_3) \\
[x \mapsto t]\mathsf{0} & = & \mathsf{0} \\
[x \mapsto t](\mathsf{succ}\ t') & = & \mathsf{succ}\ ([x \mapsto t]t') \\
[x \mapsto t](\mathsf{pred}\ t') & = & \mathsf{pred}\ ([x \mapsto t]t') \\
[x \mapsto t](\mathsf{iszero}\ t') & = & \mathsf{iszero}\ ([x \mapsto t]t')
\end{array}$$

NB: this definition is capture avoiding if we assume that t is a closed term.

Evaluation Rules

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ (\text{E-App1})$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}\ (\text{E-App2})$$

$$\frac{}{(\lambda x{:}T_{11}.\, t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}}\ (\text{E-AppAbs})$$

$$\frac{}{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow t_2}\ (\text{E-IfTrue})$$

$$\frac{}{\textsf{if false then } t_2 \textsf{ else } t_3 \longrightarrow t_3} \text{ (E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3 \longrightarrow \textsf{if } t_1' \textsf{ then } t_2 \textsf{ else } t_3} \text{ (E-If)}$$

$$\frac{t_1 \longrightarrow t_1'}{\textsf{succ } t_1 \longrightarrow \textsf{succ } t_1'} \text{ (E-Succ)}$$

$$\frac{}{\textsf{pred } 0 \longrightarrow 0} \text{ (E-PredZero)}$$

$$\frac{}{\textsf{pred } (\textsf{succ } nv_1) \longrightarrow nv_1} \text{ (E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\textsf{pred } t_1 \longrightarrow \textsf{pred } t_1'} \text{ (E-Pred)}$$

$$\frac{}{\textsf{iszero } 0 \longrightarrow \textsf{true}} \text{ (E-IsZeroZero)}$$

$$\frac{}{\textsf{iszero } (\textsf{succ } nv_1) \longrightarrow \textsf{false}} \text{ (E-IsZeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\textsf{iszero } t_1 \longrightarrow \textsf{iszero } t_1'} \text{ (E-IsZero)}$$

Typing Rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-Var)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x{:}T_1.\, t_2 : T_1 \to T_2} \text{ (T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ (T-App)}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \text{ (T-True)}$$

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}} \text{ (T-False)}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T} \text{ (T-If)}$$

$$\frac{}{\Gamma \vdash \mathsf{0} : \mathsf{Nat}} \text{ (T-Zero)}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\ t : \mathsf{Nat}} \text{ (T-Succ)}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{pred}\ t : \mathsf{Nat}} \text{ (T-Pred)}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{iszero}\ t : \mathsf{Bool}} \text{ (T-IsZero)}$$

B Reference: extending the simply typed lambda-calculus with general recursion

Extended Syntax

$$\text{Extended terms, } t \quad ::= \quad \cdots \mid \mathsf{fix}\, t \qquad \textit{fixed point of } t$$

Extended Capture-Avoiding Substitution

$$[x \mapsto t](\mathsf{fix}\, t') \quad = \quad \mathsf{fix}\,([x \mapsto t]t')$$

New Evaluation Rules

$$\frac{}{\mathsf{fix}\,(\lambda f\!:\!T_1.\, t_2) \longrightarrow [f \mapsto (\mathsf{fix}\,(\lambda f\!:\!T_1.\, t_2))]t_2}\ (\text{E-FixBeta})$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{fix}\, t_1 \longrightarrow \mathsf{fix}\, t_1'}\ (\text{E-Fix})$$

New Typing Rules

$$\frac{\Gamma \vdash t_1 : T_1 \to T_1}{\Gamma \vdash \mathsf{fix}\, t_1 : T_1}\ (\text{T-Fix})$$