CSE 130 Final, Spring 2018

Nadia Polikarpova

June 11, 2018

NAME ____

SID ____

- You have **180 minutes** to complete this exam.
- Where limits are given, **write no more** than the amount specified.
- You may refer to a **double-sided cheat sheet**, but no electronic materials.
- Questions marked with * are **difficult**; we recommend solving them last.
- Avoid seeing anyone else's work or allowing yours to be seen.
- Do not communicate with anyone but an exam proctor.
- If you have a question, raise your hand.
- **Good luck!**

Q1: Lambda Calculus: Sets [20 pts]

In this question you will implement **sets** of natural numbers in λ-calculus. Your set data structure has to support the following four operations:

```
EMPTY              -- | The empty set
INSERT n s         -- | Set that contains the number n
                   --   and all elements of the set s
HAS s n            -- | Does set s contain number n?
INTERSECT s1 s2    -- | Set that contains all the elements
                   --   common to s1 and s2
```

You can use any function defined in Appendix I (at the end of the exam). Your implementation must satisfy the following test cases:

```
let S012 = INSERT ZERO (INSERT ONE   (INSERT TWO  EMPTY))
let S234 = INSERT TWO  (INSERT THREE (INSERT FOUR EMPTY))

eval empty :
  HAS EMPTY ZERO
  =~> FALSE

eval insert_0 :
  HAS S012 ZERO
  =~> TRUE

eval insert_1 :
  HAS S012 THREE
  =~> FALSE

eval intersect_0 :
  HAS (INTERSECT S012 S234) TWO
  =~> TRUE

eval intersect_1 :
  HAS (INTERSECT S012 S234) THREE
  =~> FALSE
```

1.1 Empty set [5 pts]

```
let EMPTY = ______________________________________________________
```

1.2 Insert an element [5 pts]

```
let INSERT = _____________________________________________________
```

1.3 Membership [5 pts]

```
let HAS = ________________________________________________________
```

1.4 Set intersection [5 pts]

```
let INTERSECT = __________________________________________________
```

Q2: Datatypes and Recursion: Decision Trees [60 pts]

A **binary decision tree** (BDT) is an alternative representation of a Boolean formula. In a BDT, each *leaf* is labeled with `True` or `False`, and each *internal node* is labeled with a variable, and represents *branching* on the value of that variable.

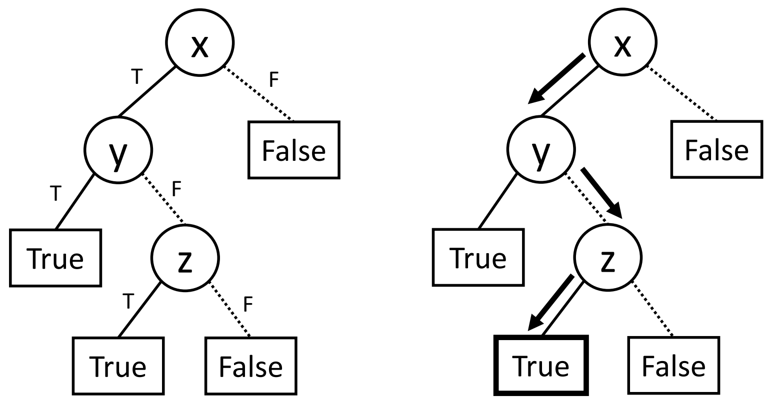

Figure 1: **(left)** A BDT representation of the formula $x \wedge (y \vee z)$. **(right)** Its evaluation with `x = True, y = False, z = True`

Figure 1 (left) shows one possible BDT representation of the Boolean formula $x \wedge (y \vee z)$. To evaluate a BDT, we start at the root; in each internal node, we descend into the left sub-tree if the node's variable is `True`, and into the right sub-tree if the node's variable is `False`); the leaf we end up in gives the value of the formula. Figure 1 (right) depicts the evaluation of our example BDT with the following variable values: `x = True, y = False, z = True`.

In this question, you will implement several Haskell functions that operate on BDTs. We will represent BDTs using the following datatype:

```haskell
data BDT = Leaf Bool | Node Id BDT BDT
```

where `Id` is just a synonym for strings:

```haskell
type Id = String
```

We will also use the type `Env` to represent an *environment*, i.e. a mapping from variable names to Boolean values:

```haskell
type Env = [(Id, Bool)]
```

Your implementation can rely on the following function to look up the value of a variable in an environment:

```haskell
lookup :: Id -> Env -> Bool
```

Besides `lookup`, your implementations can use:

- any library functions on Booleans; for example: `not`, `(&&)`, `(||)`, `(==)`
- any library functions on strings; for example: `(==)`, `(<)`, `(>)`

2.1 Evaluation [10 pts]

Implement the function `eval`, which evaluates a BDT in a given environment. You can assume that the environment contains all variables of the BDT.

Your implementation must satisfy the following test cases, where `env = [(x,True), (y,False), (z,True)]`:

```haskell
eval env (Leaf False)
  ==> False
eval env (Node "x" (Leaf True) (Leaf False))
  ==> True
eval env (Node "x" (Node "y" (Leaf True) (Leaf False)) (Leaf False))
  ==> False
```

```haskell
eval :: Env -> BDT -> Bool
```

```
---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------
```

2.2 Negation [15 pts]

The cool thing about decision trees is that you can perform logical operations (negation, conjunction, and disjunction) directly on the trees, without having to convert them back into a traditional formula representation.

Implement the function `tNot`, which returns a BDT that represents the negation of a given BDT.

Your implementation must satisfy the following test case for any BDT `t` and any environment `env`:

```
eval env (tNot t) == not (eval env t)
  ==> True
```

```
tNot :: BDT -> BDT
```

2.3 Conjunction [15 pts]

Implement the function `tAnd` that computes a conjunction of two BDTs.

Your implementation must satisfy the following test case for any two BDTs `tl` and `tr`, and any environment `env`:

```
eval env (tAnd tl tr) == (eval env tl && eval env tr)
  ==> True
```

It's okay if the resulting BDT has duplicate variables. For example, the simplest solution will satisfy the following test case (depicted on Figure 2):

```
let t = Node "x" (Leaf True) (Leaf False) in tAnd t t
  ==> Node "x" (Node "x" (Leaf True) (Leaf False)) (Leaf False)
```

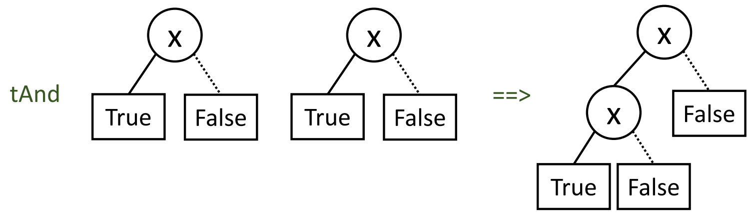

Figure 2: A test case for `tAnd`

```
tAnd :: BDT -> BDT  -> BDT
```

\---

\---

\---

\---

\---

2.4 Ordered BDTs* [20 pts]

The simple implementation of `tAnd` from section 3.3 can cause the BDT to have duplicate variables, which makes the tree less compact and slower to evaluate. One way to eliminate this redundancy is to enforce **ordering** on all the variables in the tree, such that the variable in each node is *strictly less* (lexicographically) than all variables in its sub-trees.

For example, the BDT in Figure 1 is ordered, because `x < y` and `y < z` both hold. In contrast, the BDT Figure 2 is not ordered, because `x < x` doesn't hold.

Implement the function `tAndOrd` that computes a conjunction of two *ordered* BDTs, and returns an *ordered* BDT.

Your implementation must satisfy the following test cases:

```
tAndOrd (Node "x" (Leaf True) (Leaf False))
        (Node "x" (Leaf True) (Leaf False))
  ==> (Node "x" (Leaf True) (Leaf False))

tAndOrd (Node "x" (Leaf True) (Leaf False))
        (Node "y" (Leaf True) (Leaf False))
  ==> (Node "x"
            (Node "y" (Leaf True) (Leaf False))
            (Leaf False))

tAndOrd (Node "y" (Leaf True) (Leaf False))
        (Node "x" (Leaf True) (Leaf False))
  ==> (Node "x"
            (Node "y" (Leaf True) (Leaf False))
            (Leaf False))

tAndOrd :: BDT -> BDT -> BDT

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------

---------------------------------------------------------------
```

Q3: Higher-Order Functions [40 pts]

Convert each of the given recursive functions into a function that *always returns the same result* but doesn't directly use recursion. Instead, your function can use the following higher-order functions from the standard library:

```
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldr  :: (a -> b -> b) -> b -> [a] -> b
foldl  :: (b -> a -> b) -> b -> [a] -> b
```

Apart from these four functions, your implementation can **only** use the list constructors and library functions on integers (e.g. comparisons). You are **allowed** to introduce (non-recursive) auxiliary functions.

3.1 List reversal [5 pts]

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Non-recursive version:

```
reverse :: [a] -> [a]


----------------------------------------------------------------

----------------------------------------------------------------
```

3.2 Absolute values [10 pts]

```
absValues :: [Int] -> [Int]
absValues []     = []
absValues (x:xs)
  | x < 0      = (-x):(absValues xs)
  | otherwise  =   x :(absValues xs)
```

Non-recursive version:

```
absValues :: [Int] -> [Int]


--------------------------------------------------------------

--------------------------------------------------------------
```

3.3 Remove duplicates [15 pts]

```
dedup :: [Int] -> [Int]
dedup [] = []
dedup (x:xs) = x:(remove x (dedup xs))
  where
    remove x [] = []
    remove x (y:ys)
      | x == y    =    remove x ys
      | otherwise = y:(remove x ys)
```

Non-recursive version:

```
dedup :: [Int] -> [Int]


--------------------------------------------------------------

--------------------------------------------------------------

--------------------------------------------------------------

--------------------------------------------------------------
```

3.4 Insertion Sort* [20 pts]

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) = insert x (sort xs)
  where
    insert x []     = [x]
    insert x (y:ys) = if x <= y
                        then x:y:ys
                        else y:(insert x ys)
```

Non-recursive version:

```
sort :: [Int] -> [Int]

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------
```

Q4: Semantics and Type Systems [30 pts]

In this question you will use the operational semantics and the type system of Nano2 (given in Appendix II at the end of the exam) to derive some reduction judgments `E,e => E',e'` and typing judgments `G |- e :: S`.

A *complete derivation* should satisfy the following conditions:

- all judgments in the derivation are complete
- every rule (or axiom) application is labeled with the name of the rule
- all leaves are axioms

Here is an example of a complete reduction derivation:

```
                [Add] ----------------
                           E, 1 + 2 => E, 3
[Add-L] ----------------------------------------
        E, (1 + 2) + (4 + 5)  =>  E, 3 + (4 + 5)
```

4.1 Reduction 1 [10 points]

Complete the following reduction derivation, where

`E = [f -> <[], \x y -> x + y>]`

```
[______] -------------------------------------------------------

         E,           => E,

[______] -------------------------------------------------------

         E,           => E,

[______] -------------------------------------------------------

         E, f 1 2     => E,
```

4.2 Reduction 2 [10 points]

Complete the following reduction derivation (same `E` as in 4.1):

```
[______] -------------------------------------------------------

                                   =>

[______] -------------------------------------------------------

         E, <[], \x y -> x + y> 1 2 =>
```

4.3 Typing 1 [10 points]

Complete the following typing derivation

```
[______]----------------------       -----------------------[______]


[______]-------------------------------------------------


[______]-------------------------------------------------

        [] |- \x -> x + 5 ::
```

4.4 Typing 2 [10 points]

Complete the following typing derivation where

`G = [f : Int -> Int, id : forall a. a -> a]`

```
[______]------------------------------

        G |-

[______]------------------------------     --------------------[______]

        G |-                                G |-

[______]---------------------------------------------------------------

        G |- id f ::
```

Q5: Prolog: Selection sort [30 pts]

In this question, we will implement Selection sort in Prolog. As a reminder, this algorithm sorts a list by repeatedly extracting the *minimum element* from the input list and appending it to the output list.

Unless otherwise stated, your solution **cannot** use any library functions/predicates or introduce auxiliary predicates.

5.1 Insert [10 points]

Write a Prolog predicate `insert(X, Ys, Zs)` which is true whenever `Zs` is the result of inserting the element `X` into `Ys` at some position.

Your implementation should satisfy the following test cases

```
?- insert(1, [2,3], Zs).
Zs = [1,2,3];
Zs = [2,1,3];
Zs = [2,3,1];
false.

?- insert(1, Ys, [1,2,1]).
Ys = [2,1];
Ys = [1,2];
false.
```

5.2 Minimum element [10 points]

Write a Prolog predicate `list_min(Acc, Xs, Min)` which is true when `Min` is the minimum between the number `Acc` and the smallest element of list `Xs` (if non-empty).

In this problem, you can use the built-in function `min(X,Y)` that computes the minimum of two numbers.

Your implementation should satisfy the following test cases

```
?- list_min(1, [], X).
X = 1; false.

?- list_min(1, [3,2], X).
X = 1; false.

?- list_min(2, [3,1], X).
X = 1; false.
```

```
-----------------------------------------------------------------

-----------------------------------------------------------------

-----------------------------------------------------------------

-----------------------------------------------------------------

-----------------------------------------------------------------
```

5.3 Selection Sort [10 points]

Write a Prolog predicate `selection_sort(Xs, Ys)` which is true when the list `Ys` contains the same elements as `Xs` but in ascending order.

Your solution can use the predicates `insert` and `list_min` defined in 5.1 and 5.2.

Your implementation should satisfy the following test cases (when queried for the first solution only).

```
?- selection_sort([3,2,4,1], Ys).
Ys = [1,2,3,4].

?- selection_sort([1,2,1,2], Ys).
Ys = [1,1,2,2].
```

```
------------------------------------------------------------

------------------------------------------------------------

------------------------------------------------------------

------------------------------------------------------------

------------------------------------------------------------
```

Appendix I: Lambda Calculus Cheat Sheet

Here is a list of definitions you may find useful for Q2

```
-- Booleans ---------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2

-- Pairs ------------------------------------

let PAIR  = \x y b -> b x y
let FST   = \p     -> p TRUE
let SND   = \p     -> p FALSE

-- Numbers ----------------------------------

let ZERO  = \f x -> x
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let THREE = \f x -> f (f (f x))

-- Arithmetic -------------------------------

let INC   = \n f x -> f (n f x)
let ADD   = \n m -> n INC m
let MUL   = \n m -> n (ADD m) ZERO
let ISZ   = \n -> n (\z -> FALSE) TRUE
let EQL   = \n m -> AND (ISZ (SUB n m)) (ISZ (SUB m n))
```

Appendix II: Syntax and Semantics of Nano2

Expression syntax:

```
e ::= n | x | e1 + e2 | let x = e1 in e2 | \x -> e | e1 e2
```

Operational semantics:

```
[Var]       E, x => E, E[x]          if x in dom(E)

[Add]       E, n1 + n2 => E, n       where n == n1 + n2

                 E, e1 => E', e1'
[Add-L]     --------------------------
            E, e1 + e2 => E', e1' + e2

                 E, e2 => E', e2'
[Add-R]     --------------------------
            E, n1 + e2 => E', n1 + e2'

[Let]       E, let x = v in e2 => E[x->v], e2

                          E, e1 => E', e1'
[Let-Def]   --------------------------------------------
            E, let x = e1 in e2 => E', let x = e1' in e2

[Abs]       E, \x -> e => E, <E,\x -> e>

[App]       E, <E1,\x -> e> v => E1[x->v], e

               E, e1 => E', e1'
[App-L]     ----------------------
            E, e1 e2 => E', e1' e2

              E, e => E', e'
[App-R]     ------------------
            E, v e => E', v e'
```

Syntax of types:

```
T ::= Int | T1 -> T2 | a
S ::= T | forall a . S
```

Typing rules:

```
[T-Num]  G |- n :: Int

         G |- e1 :: Int   G |- e2 :: Int
[T-Add]  -------------------------------
               G |- e1 + e2 :: Int

[T-Var]  G |- x :: S          if x:S in G

           G, x:T1 |- e :: T2
[T-Abs]  ------------------------
         G |- \x -> e :: T1 -> T2

         G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App]  -----------------------------------
                  G |- e1 e2 :: T2

         G |- e1 :: S   G, x:S |- e2 :: T
[T-Let]  --------------------------------
            G |- let x = e1 in e2 :: T

         G |- e :: forall a . S
[T-Inst] ----------------------
          G |- e :: [a / T] S

               G |- e :: S
[T-Gen]  ---------------------- if not (a in FTV(G))
         G |- e :: forall a . S
```

Here $n \in \mathbb{N}$ is natural number, $v \in \text{Val}$ is a value, $x \in \text{Var}$ is a variable, $e \in \text{Expr}$ is an expression, $E \in \text{Var} \to \text{Val}$ is an environment, $a \in \text{TVar}$ is a type variable, $T \in \text{Type}$ is a type, $S \in \text{Poly}$ is a type scheme (a poly-type), $G \in \text{Var} \to \text{Poly}$ is a type environment (a context).