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:

```lambda
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 = ________________________________`
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.

![Decision Tree Diagram](image)

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

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:
type Env = [(Id, Bool)]

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

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)]:

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

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`:

\[
\text{eval env (tNot t)} = \text{not (eval env t)}
\]

\[
\text{tNot :: BDT -> BDT}
\]

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`:

\[
\text{eval env (tAnd tl tr)} = \text{eval env tl} \land \text{eval env tr}
\]

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

\[
\text{let t = Node "x" (Leaf True) (Leaf False) in tAnd t t}
\]

\[
\Rightarrow \text{Node "x" (Node "x" (Leaf True) (Leaf False)) (Leaf False)}
\]
The simple implementation of \texttt{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 \texttt{tAndOrd} that computes a conjunction of two ordered BDTs, and returns an ordered BDT.

Your implementation must satisfy the following test cases:

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node (tAnd) at (0,0) {$\wedge$};
\node[below] at (tAnd) {true \ false};
\node[below] at (tAnd |- tAnd) {true \ false};
\node[below] at (tAnd |- tAnd) {$\Rightarrow$};
\node[below] at (tAnd |- tAnd) {true \ false};
\end{tikzpicture}
\caption{A test case for \texttt{tAnd}}
\end{figure}
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))

\textit{tAndOrd} :: \textit{BDT} -> \textit{BDT} -> \textit{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]

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

Non-recursive version:

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

### 3.2 Absolute values [10 pts]

```haskell
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]

\[
\text{sort} :: [\text{Int}] \rightarrow [\text{Int}]
\]

\[
\text{sort} \; [] = []
\]

\[
\text{sort} \; (x:xs) = \text{insert} \; x \; (\text{sort} \; xs)
\]

where

\[
\text{insert} \; x \; [] = [x]
\]

\[
\text{insert} \; x \; (y:ys) = \begin{cases} 
  x:y:ys & \text{if } x \leq y \\
  y:(\text{insert} \; x \; ys) & \text{else}
\end{cases}
\]

Non-recursive version:

\[
\text{sort} :: [\text{Int}] \rightarrow [\text{Int}]
\]

\[
\begin{multline*}
\text{---------------------------------------------}\\
\text{---------------------------------------------}\\
\text{---------------------------------------------}\\
\text{---------------------------------------------}\\
\text{---------------------------------------------}\\
\text{---------------------------------------------}\\
\end{multline*}
\]
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 $E, e \Rightarrow E', e'$ and typing judgments $G \vdash e :: S$.

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

4.1 Reduction 1 [10 points]

Complete the following reduction derivation, where

$E = [f \rightarrow <[]>, \lambda x \ y \rightarrow x + y]$  

```
[______] -------------------------------------------------------
E, \Rightarrow E,
[______] -------------------------------------------------------
E, \Rightarrow E,
[______] -------------------------------------------------------
E, f 1 2 \Rightarrow E,
```
4.2 Reduction 2 [10 points]

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

\[ \text{[______]} \quad \text{--------------------------------------------------------} \]

\[ \Rightarrow \]

\[ \text{[______]} \quad \text{--------------------------------------------------------} \]

\[ \text{E, } <[], \lambda x \ y \rightarrow x + y> \ 1 \ 2 \Rightarrow \]

4.3 Typing 1 [10 points]

Complete the following typing derivation

\[ \text{[______]} \quad \text{---------------------- -----------------------[______]} \]

\[ \text{[______]} \quad \text{-------------------------------------------------} \]

\[ \text{[______]} \quad \text{-------------------------------------------------} \]

\[ \text{[______]} \quad \text{[______]} \]

\[ \text{E, }[] \ |- \ \lambda x \rightarrow x + 5 : : \]
4.4 Typing 2 [10 points]

Complete the following typing derivation where 
G = [f : Int -> Int, id : forall a. a -> a]

\[
\begin{align*}
G & \vdash \\
\text{[_____]-----------------------------} \\
G & \vdash \\
\text{[_____]--------------------------------} & \text{[_____]}
\end{align*}
\]

\[
\begin{align*}
G & \vdash \\
\text{[_____]--------------------------------} & \text{[_____]}
\end{align*}
\]

\[
\begin{align*}
G & \vdash \\
G & \vdash \\
\text{[_____]---------------------------------------------}
\end{align*}
\]

\[
\begin{align*}
G & \vdash \text{id f ::}
\end{align*}
\]
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

```prolog
?- 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:

\[ e ::= n \mid x \mid e_1 + e_2 \mid \text{let } x = e_1 \text{ in } e_2 \mid \backslash x \to e \mid e_1 e_2 \]

Operational semantics:

- **[Var]** \( E, x \Rightarrow E, E[x] \) if \( x \in \text{dom}(E) \)
- **[Add]** \( E, n_1 + n_2 \Rightarrow E, n \) where \( n = n_1 + n_2 \)
- **[Add-L]** \( E, e_1 + e_2 \Rightarrow E, e_1' + e_2 \)
- **[Add-R]** \( E, n_1 + e_2 \Rightarrow E, n_1 + e_2' \)
- **[Let]** \( E, \text{let } x = v \text{ in } e_2 \Rightarrow E[x\to v], e_2 \)
- **[Let-Def]** \( E, \text{let } x = e_1 \text{ in } e_2 \Rightarrow E, \text{let } x = e_1' \text{ in } e_2 \)
- **[Abs]** \( E, \backslash x \to e \Rightarrow E, \langle E, \backslash x \to e \rangle \)
- **[App]** \( E, \langle E_1, \backslash x \to e \rangle v \Rightarrow E_1[x\to v], e \)
- **[App-L]** \( E, e_1 e_2 \Rightarrow E', e_1' e_2 \)
- **[App-R]** \( E, v e \Rightarrow E', v e' \)
Syntax of types:

\[ T ::= \text{Int} \mid T_1 \rightarrow T_2 \mid a \]

\[ S ::= T \mid \forall a . S \]

Typing rules:

\[ [T-\text{Num}] \quad \Gamma \vdash n :: \text{Int} \]

\[ \Gamma \vdash e_1 :: \text{Int} \quad \Gamma \vdash e_2 :: \text{Int} \]

\[ \text{-----------------------------} \]

\[ \Gamma \vdash e_1 + e_2 :: \text{Int} \]

\[ [T-\text{Var}] \quad \Gamma \vdash x :: S \quad \text{if } x : S \text{ in } \Gamma \]

\[ \Gamma, x : T_1 \vdash e :: T_2 \]

\[ [T-\text{Abs}] \quad \text{------------------------} \]

\[ \Gamma \vdash \lambda x \to e :: T_1 \rightarrow T_2 \]

\[ \Gamma \vdash e_1 :: T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 :: T_1 \]

\[ \text{-----------------------------------} \]

\[ \Gamma \vdash e_1 \ e_2 :: \text{T}_2 \]

\[ [T-\text{App}] \quad \Gamma \vdash e_1 :: S \quad \Gamma, x : S \vdash e_2 :: T \]

\[ \text{----------------------------------------} \]

\[ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 :: T \]

\[ [T-\text{Let}] \quad \text{---------------------------------} \]

\[ \Gamma \vdash e :: \forall a . S \]

\[ [T-\text{Inst}] \quad \text{------------------------} \]

\[ \Gamma \vdash e :: [a / T] S \]

\[ [T-\text{Gen}] \quad \text{------------------------} \] \text{if not (} a \text{ in } \text{FTV}(\Gamma) \text{)}

\[ \Gamma \vdash e :: \forall a . S \]

Here \( n \in \mathbb{N} \) is a 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), \( \Gamma \in \text{Var} \to \text{Poly} \) is a type environment (a context).