CSE 130 Final, Winter 2019

Nadia Polikarpova

March 22, 2019

NAME _____

SID _____

- DO NOT TURN THIS PAGE OVER BEFORE WE TELL YOU TO
- 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.
- 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 [25 pts]

Below you will find *five* lambda terms, `F1` through `F5` (along with helper functions, `H1` and `H2`). Each one of them implements one of *ten* possible functions, `(A)` through `(J)`. Your task is to guess which term implements which function.

Fill in the table below so that each *column* has exactly one mark, and each *row* has at most one mark. You get 5 points for each lambda term whose meaning you guessed correctly. You will find the definitions of all the functions we used in Appendix I at the end of the exam.

```
let F1 = \n   -> n NOT TRUE

let F2 = \n   -> n (MUL TWO) ONE

let F3 = \x y -> ISZ (SUB y x) x y

let H1 = \p   -> PAIR (NOT (FST p)) (ITE (FST p) (INC (SND p)) (SND p))
let F4 = \n   -> SND (n H1 (PAIR FALSE ZERO))

let H2 = \p   -> PAIR (INC (FST p)) (ADD (FST p) (SND p))
let F5 = \n   -> SND (n H2 (PAIR ONE ZERO))
```

	F1	F2	F3	F4	F5	
(A) max of x and y	[ ]	[ ]	[ ]	[ ]	[ ]	(A)
(B) x < y	[ ]	[ ]	[ ]	[ ]	[ ]	(B)
(C) x > y	[ ]	[ ]	[ ]	[ ]	[ ]	(C)
(D) n squared	[ ]	[ ]	[ ]	[ ]	[ ]	(D)
(E) 2 to the power of n	[ ]	[ ]	[ ]	[ ]	[ ]	(E)
(F) n divided by 2	[ ]	[ ]	[ ]	[ ]	[ ]	(F)
(G) is n even?	[ ]	[ ]	[ ]	[ ]	[ ]	(G)
(H) constant false	[ ]	[ ]	[ ]	[ ]	[ ]	(H)
(I) n-th fibonacci	[ ]	[ ]	[ ]	[ ]	[ ]	(I)
(J) sum from 0 to n	[ ]	[ ]	[ ]	[ ]	[ ]	(J)

Q2: Haskell: Files and Directories [35 pts]

Recall the Haskell representation of files and directories from the midterm. We can represent a directory structure using the following Haskell datatype:

```haskell
data Entry =
    File String Int      -- file: name and size
  | Dir String [Entry]   -- directory: name and child entries
```

For example, the value:

```haskell
homedir = Dir "home"
            [ File "todo" 256
            , Dir  "HW0" [ File "Makefile" 575 ]
            , Dir  "HW1" [ File "Makefile" 845, File "HW1.hs" 3007]
            ]
```

represents the following directory structure:

```
home
 |---todo (256 bytes)
 |
 |---HW0
 |     |---Makefile (575  bytes)
 |
 |---HW1
       |---Makefile (845  bytes)
       |---HW1.hs   (3007 bytes)
```

In your solutions you can use any library functions on integers (e.g. arithmetic operators), but **only the following** functions on lists:

```haskell
(==)   :: String -> String -> Bool       -- equality on strings
(++)   :: [a] -> [a] -> [a]              -- append on any lists
```

2.1 Tail-Recursive Size [15 pts]

Recall the function `size` that computes the total size of an entry in bytes. Implement a *tail-recursive* version of this function, using a helper function `loop` with the signature provided below. *Hint:* the second argument of loop is a list of entries yet to be processed.

Your implementation must satisfy the following test cases

```
size (File "todo" 256)
  ==> 256
size (Dir "haskell-jokes" [])
  ==> 0
size homedir
  ==> 4683    -- 256 + 575 + 845 + 3007
```

```
size :: Entry -> Int

size e = ____________________________________________________

  where

    loop :: Int -> [Entry] -> Int

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________
```

2.2 Remove [15 pts]

Implement the *higher-order* function `remove p e`, which recursively traverses all the sub-entries inside `e` and removes those that satisfy the predicate `p`.

Your implementation must satisfy the following test cases (here `nameOf` is a function that returns the name of an entry):

```
remove (\e -> nameOf e == "Makefile") homedir
  ==> Dir "home"
            [ File "todo" 256
            , Dir  "HW0" []
            , Dir  "HW1" [File "HW1.hs" 3007]
            ]

remove (\e -> nameOf e == "HW1") homedir
  ==> Dir "home"
            [ File "todo" 256
            , Dir  "HW0" [ File "Makefile" 575 ]
            ]

remove (\e -> nameOf e == "home") homedir
  -- the current directory is never removed (only sub-entries),
  -- so return homedir unchanged:
  ==> Dir "home"
            [ File "todo" 256
            , Dir  "HW0" [ File "Makefile" 575 ]
            , Dir  "HW1" [ File "Makefile" 845, File "HW1.hs" 3007]
            ]
```

```
remove :: (Entry -> Bool) -> Entry -> Entry
```

2.3 Clean up [5 pts]

Using `remove` from 2.2, implement the function `cleanup e` that removes all empty subdirectories of `e`. Your implementation must satisfy the following test cases:

```
cleanup (Dir "temp" [Dir "drafts" [], File "todo" 256])
  ==> Dir "temp" [File "todo" 256]
cleanup (File "todo" 256)
  ==> File "todo" 256
cleanup (Dir "drafts" [])
  -- the current directory is never removed (only sub-dirs):
  ==> Dir "drafts" []
```

```
cleanup :: Entry -> Entry
```

Q3: Semantics and Type Systems [20 pts]

In this part, you will answer questions about the operational semantics and the type system of Nano2, both given in Appendix II at the end of the exam. In each question, mark **all** the answers that apply; it is possible that *none*, *some*, or *all* of the answers are correct.

3.1 Reduction 1 [5 points]

Which of these single-step reductions are valid according to the operational semantics of Nano2?

(A) `5 => 5` [ ]

(B) `(\x -> x) (1 + 2) => (\x -> x) 3` [ ]

(C) `(\x -> x) (1 + 2) => 1 + 2` [ ]

(D) `(\x -> x) (1 + 2) => 3` [ ]

(E) `(1 + 2) + (\x -> x) => 3 + (\x -> x)` [ ]

3.2 Reduction 2 [5 points]

Which of the following rules are used in the derivation of the reduction

`(\x y -> (x + y) + (1 + 2)) (3 + 4) 5 => ???`

(A) `Add-L` [ ]

(B) `Add-R` [ ]

(C) `Add` [ ]

(D) `App-L` [ ]

(E) `App-R` [ ]

3.3 Typing 1 [5 points]

Which of the following typing judgments are valid according to the type system of Nano2?

(A) `[] |- \x -> x :: Int -> Int` [ ]

(B) `[] |- \x -> x :: a -> a` [ ]

(C) `[] |- \x -> x :: forall a . a -> a` [ ]

(D) `[] |- x :: Int` [ ]

(E) `[x: a] |- x :: forall a . a` [ ]

3.4 Typing 2 [5 points]

Which of the following rules are used in the derivation of the typing judgment

```
[] |- \x y -> x y :: forall a . forall b . (a -> b) -> a -> b
```

(A) `T-Var` [ ]

(B) `T-Abs` [ ]

(C) `T-App` [ ]

(D) `T-Inst` [ ]

(E) `T-Gen` [ ]

Q4: Prolog: Regular expressions [30 pts]

In this question, we will implement *regular expression* matching in Prolog. More precisely, your task is to define a predicate `match(R, S)` where `R` is a term that describes the regular expression and `S` is a list (which can contain atoms and numbers).

We will represent regular expressions using the following terms:

- `oneOf(Xs)` matches a singleton list whose element is one of the elements of the list `Xs`
- `seq(R1,R2)` matches any list that can be split into a prefix and a suffix, where the prefix matches `R1` and the suffix matches `R2`
- `star(R)` matches zero or more repetitions of `R`

Once you are done, your implementation should pass the following tests:

```
match(oneOf([0,1,2,3]), [1]).
true.

match(oneOf([0,1,2,3]), []).
false.

match(oneOf([0,1,2,3]), [1,3,0]).
false.

match(seq(oneOf([0,1,2,3]), oneOf([0,1,2,3])), [1,3]).
true.

match(seq(oneOf([0,1,2,3]), oneOf([0,1,2,3])), [1]).
false.

match(seq(oneOf([0,1,2,3]), oneOf([0,1,2,3])), [c,1]).
false.

match(star(oneOf([0,1,2,3])), [1,3,0]).
true.
```

```
match(star(oneOf([0,1,2,3])), []).
true.

match(star(oneOf([0,1,2,3])), [c,s,e,1,3,0]).
false.

match(seq(oneOf([c]), seq(oneOf([s]), seq(oneOf([e]),
  star(oneOf([0,1,2,3]))))), [c,s,e,1,3,0]).
true.

match(seq(oneOf([c]), seq(oneOf([s]), seq(oneOf([e]),
  star(oneOf([0,1,2,3]))))), [c,s,1,3,0]).
false.
```

In each question below, you will define **one or more rules** for the `match` predicate for each of the three kinds of regular expressions. You can use patterns in the head of the rule, but **cannot** introduce auxiliary predicates. The **only** library predicate you can use is `append(Xs,Ys,Zs)`, which is true when the list `Zs` is the result of appending `Ys` to `Xs`.

4.1 One of [10 points]

Define the rule(s) for `match(oneOf(Xs), S)`:

4.2 Sequential Composition [10 points]

Define the rule(s) for `match(seq(R1, R2), S)`:

4.3 Kleene Star [10 points]

Define the rule(s) for `match(star(R), S)`:

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

-- 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 SKIP1 = \f p -> PAIR TRUE (ITE (FST p) (f (SND p)) (SND p))
let DEC   = \n   -> SND (n (SKIP1 INC) (PAIR FALSE ZERO))
let SUB   = \n m -> m DEC n
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:

```
                e1  =>  e1'
[Add-L]   ---------------------
          e1 + e2  =>  e1' + e2

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

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

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

[Let]     let x = v in e2  =>  e2[x := v]

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

            e  =>  e'
[App-R]   -------------
          v e  =>  v e'

[App]     (\x -> e) v  =>  e[x := v]
```

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, $v \in$ Val is a value, $x \in$ Var is a variable, $e \in$ Expr is an expression, $a \in$ TVar is a type variable, $T \in$ Type is a type, $S \in$ Poly is a type scheme (a poly-type), $G \in$ Var $\to$ Poly is a type environment (a context).