

CPSC 449 (Fall 2020)  
Programming Paradigms  
Final Exam

Instructor: Dr. Philip W. L. Fong

Starting Time: 6:30 PM, Wednesday, December 9, 2020

Ending Time: 6:30 PM, Saturday, December 12, 2020

Consult the D2L site for detailed instructions and policies regarding this exam.

1. [10%] A **binary relation** over `Integer` can be represented as a list of `Integer` pairs.

```
type BinRel = [(Integer, Integer)]
```

Given two binary relations  $R_1$  and  $R_2$  represented as above, the composition  $R_1 \circ R_2$  is a list containing those pairs  $(x, z)$  for which there exists an `Integer`  $y$  such that  $(x, y)$  is a member of the list  $R_1$  and  $(y, z)$  is a member of the list  $R_2$ . Develop the following Haskell function, which computes the composition of two `BinRels`:

```
bin_rel_comp :: BinRel -> BinRel -> BinRel
```

For example:

```
bin_rel_comp [(1,2), (7,9)]  
              [(2,3), (3,4), (9,1), (9,4)]  
  ~> [(1,3), (7,1), (7,4)]
```

Your solution shall *NOT* involve explicit recursion. Use list comprehension instead. A solution involving explicit recursion will receive zero mark.

2. [12%] A **bit vector** is represented as a list of boolean values in Haskell.

```
type BitVec = [Bool]
```

The binary representation of a non-negative integer can be encoded as a `BitVec`, such that the head of the list is the least significant bit. For example, the binary representation of the number 12 is  $1100_2$ , and the latter is encoded as the `BitVec` below:

```
[False, False, True, True]
```

Note that the bits are listed from the least significant to the most significant.

Develop a Haskell function:

```
addbv :: BitVec -> BitVec -> BitVec
```

such that, if `xs` and `ys` are the `BitVec` encodings of non-negative integers  $X$  and  $Y$ , then `(addbv xs ys)` is the `BitVec` encoding of  $X + Y$ . For example:

```

(addbv [True, True] [False, True, False, True])
  ~> [True, False, True, True]
(addbv [] [True, True])
  ~> [True, True]
(addbv [True, True] [])
  ~> [True, True]
(addbv [] [])
  ~> []

```

You are expected to demonstrate your ability to formulate recursive functions. **Hint:** Introduce helper functions to help you break the problem into manageable subproblems.

3. [12%] Consider the following function.

```

crunch :: [Integer] -> [Integer] -> Integer
crunch [] [] = 0 (crunch.1)
crunch (x:xs) [] = crunch xs [x] (crunch.2)
crunch [] (y:ys) = y + crunch [] ys (crunch.3)
crunch (x:xs) (y:ys)
  | even x = crunch xs (x:y:ys) (crunch.4)
  | otherwise = crunch ((x+y):xs) ys (crunch.5)

```

(The source code above can also be found in the file `SampleCode.hs`.)

- (a) [1%] Give a conjecture of what the `crunch` function returns. **Hint:** Use equational reasoning to help yourself understand how recursion unfolds in `crunch`. There is, however, no need to include such equational reasoning work with your submission. Only report your conjecture.
  - (b) [4%] Propose a rank function that can be used for demonstrating the termination of `crunch`. More specifically, formulate a function `rank(xs,ys)` that takes two lists of `Integers` as input, and returns a natural number as output. **Hint:** You are asked to formulate a mathematical function. There is no need to write any Haskell code.
  - (c) [7%] Use the rank function you formulated above to prove that `crunch` always terminates when given finite lists as input.
4. [10%] The following algebraic type is defined for capturing the abstract syntax of a simple expression language.

```

type VarName = Char

data Expr = Lit Integer
          | Var VarName
          | Add Expr Expr

```

In short, an expression can be an integer literal, a variable (whose name is a single character), or the sum of two expressions. For example, the expression  $(x + 3) + y$  is represented as follows:

```
ex :: Expr
ex = Add (Add (Var 'x') (Lit 3)) (Var 'y')
```

A *substitution* represents a mapping of variable names to expressions.

```
type Binding = (VarName, Expr)
type Substitution = [Binding]
```

In short, a substitution is encoded as a list of bindings, each of which is a pair containing a variable name and an expression. The following substitution maps variable  $x$  to the literal 7, variable  $y$  to variable  $z$ , and variable  $z$  to the literal 0.

```
sub :: Substitution
sub = [('x', Lit 7), ('y', Var 'z'), ('z', Lit 0)]
```

Develop a Haskell function `substitute`, which takes a substitution `s` and an expression `e` as arguments, and returns an expression obtained by applying `s` to `e`. The following is the type signature of the function.

```
substitute :: Substitution -> Expr -> Expr
```

The following is an example of the output of the function when the sample expression and substitution above are given as arguments.

```
substitute sub ex
~> (Add (Add (Lit 7) (Lit 3)) (Var 'z'))
```

**Hint:** Note that a typical implementation of Haskell does not know how to print out an expression or a substitution, as they are not built-in types. To help you debug your code, two helper functions `showExpr` and `showSub` have been provided to you in `SampleCode.hs`. They can be used respectively for converting expressions and substitutions to strings, so you can “print” them out for debugging purposes.

5. [12%] Recall the simple expression language above.

```
type VarName = Char

data Expr = Lit Integer
          | Var VarName
          | Add Expr Expr
```

The function `numVars` returns the number of variable occurrences in a given expression.

```

numVars :: Expr -> Integer
numVars (Lit _) = 0                                (numVars.1)
numVars (Var _) = 1                                (numVars.2)
numVars (Add e1 e2) = (numVars e1) + (numVars e2)  (numVars.3)

```

The function `leftHeight` returns the number of `Add` appearing in the leftmost branch of a given expression.

```

leftHeight :: Expr -> Integer
leftHeight (Lit _) = 0                                (leftHeight.1)
leftHeight (Var _) = 0                                (leftHeight.2)
leftHeight (Add e _) = 1 + (leftHeight e)             (leftHeight.3)

```

The function `size` returns the number of constructors used for constructing a given expression.

```

size :: Expr -> Integer
size (Lit _) = 1                                (size.1)
size (Var _) = 1                                (size.2)
size (Add e1 e2) = 1 + (size e1) + (size e2)        (size.3)

```

Use structural induction to prove that, for every finite expression `e`,

$$(\text{numVars } e) + (\text{leftHeight } e) \leq \text{size } e$$

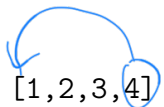
The presentation of your proof shall conform to the following format. Deviation will lead to significant mark reduction.

- (a) [2%] State the Principle of Structural Induction for the algebraic type `Expr`.
  - (b) [2%] State the concrete proof goals.
  - (c) [8%] Prove the proof goals.
6. [12%] Use `map`, `filter`, and/or `foldr/foldr1` to implement the following Haskell functions. You shall demonstrate the use of anonymous functions (i.e., lambda abstractions) in the implementation of at least one of following functions. Failing to demonstrate the use of anonymous functions will lead to a 3% mark deduction.
- (a) [4%] `sum_sq_even :: [Integer] -> Integer`. The function returns the sum of the squares of the even members of the input list. For example:
 

```

sum_sq_even [1, 2, 3, 4] ~ 20
sum_sq_even [1, 3] ~ 0
sum_sq_even [] ~ 0

```
  - (b) [4%] `bubble_up :: [Integer] -> [Integer]`. The function returns the a list almost identical to the input list, except that the last element of the input list is now the first element of the returned list. For example:


  
 bubble\_up [1,2,3,4]  $\rightsquigarrow$  [4,1,2,3]
   
 bubble\_up [4,1,2,3]  $\rightsquigarrow$  [3,4,1,2]
   
 bubble\_up [1]  $\rightsquigarrow$  [1]
   
 bubble\_up []  $\rightsquigarrow$  []

- (c) [4%] `boost_all :: [Integer->Integer] -> (Integer->Integer)`. The requirement of `boost_all` is specified below.

Given an integer function  $f : \text{Integer} \rightarrow \text{Integer}$ , the **boost** of  $f$  is the integer function  $f^\uparrow : \text{Integer} \rightarrow \text{Integer}$  defined as follows:

$$f^\uparrow(n) = \begin{cases} f(n) & \text{if } f(n) > n \\ n & \text{otherwise} \end{cases}$$

$f\text{-up} :: n =$   
 $\text{if } f(n) > n \text{ then } f(n) \text{ else } n$

The function `boost_all` takes a non-empty list of integer functions as input

$$L = [f_1, f_2, \dots, f_k] \quad [f_i \mid f_i \leftarrow L]$$

and returns a function obtained by composing the boosts of the input functions

$$f_1^\uparrow \circ f_2^\uparrow \circ \dots \circ f_k^\uparrow$$

7. [12%] A **bit vector** is represented as a list of boolean values in Haskell.

```
type BitVec = [Bool]
```

A bit vector `bv` is said to be **elegant** if and only if (a) the vector is identical to its reversal (i.e., `bv == reverse bv`), and (b) the length of the bit vector is an odd number (i.e., `odd (length bv)`).

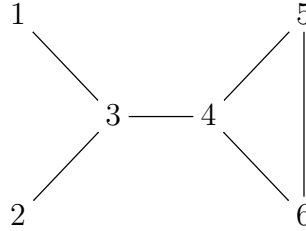
Create an infinite list, `elegant :: [BitVec]`, that contains all bit vectors that are elegant. More precisely, a bit vector `bv` is elegant if and only if the Haskell function call `(member bv elegant)` returns `True` in finite time. In other words, every elegant bit vector appears at least once in `elegant`, while none of the inelegant bit vectors appears in `elegant`.

8. [20%] An indoor space is modelled as an undirected graph  $G = (V, E)$ . Every vertex indicates a standing spot (e.g., where you stand when you line up in a grocery store). Two vertices are adjacent to one another if they are considered to be in close proximity. If two persons are standing at the spots represented by adjacent vertices, then they fail to observe social distancing. A question worth asking is the following: Is it possible to have  $k$  people entering this indoor space, so that they stand at appropriate spots while observing social distancing (i.e., no two persons stand at adjacent spots)? To rephrase this question in mathematical terms, given an undirected graph  $G = (V, E)$ , is it possible to select a subset  $S$  of  $k$  vertices from  $V$ , so that no two vertices in  $S$  are adjacent to one another? Such a vertex set  $S$  is called a **socially distant set**. Your goal is to devise a Prolog program to check for the existence of socially distant sets for a given  $k$ .

An undirected graph in Prolog is represented by a list of pairs, so that each pair encoding an *undirected* edge. As an example, consider the following Prolog list:

[(1,3), (2,3), (3,4), (4,5), (4,6), (5,6)]

This list represents the graph below:



We can place 3 persons on vertex 1, 2, and 4, while maintaining social distancing. Here  $k$  is 3, and the socially distant set  $S$  can be represented by the following list:

[1, 2, 4]

With the existence of this  $S$ , we know that a socially distant set exists for  $k = 3$ . In fact, 3 is the biggest  $k$  for which a socially distant set exists for the undirected graph above. For example, if we pick  $S$  to be the following set, we will fail to maintain social distancing, since vertex 5 and vertex 6 are adjacent to one another.

[1, 2, 5, 6]

Develop the following Prolog predicates.

- (a) [5%] Develop a Prolog predicate, `check_soc_dist(+G, +S)`, which asserts that the list  $S$  is a socially distant set for the undirected graph  $G$ . Note that both  $G$  and  $S$  are assumed to be fully instantiated. This part is to get you familiarize with the definition of socially distant sets.
- (b) [5%] Develop a Prolog predicate, `collect_vertices(+G, -Vs)`, which asserts that the list  $Vs$  is the list of all vertices in the undirected graph  $G$ . The argument  $Vs$  is supposed to be a variable when the predicate is called. On return, the variable will be instantiated to a list of vertices appearing in the list  $G$  of edges. The predicate will generate only one possible value for  $Vs$ . **Hint:** You may use this predicate as a helper in the following part.
- (c) [10%] Develop a Prolog predicate, `gen_soc_dist(+G, -S)`, which asserts that the list  $S$  is a socially distant set for the undirected graph  $G$ . Note that, unlike part (a) above, this time  $S$  is a variable when the predicate is called. The predicate will generate all possible socially distant sets in turn. For example, invoking `gen_soc_dist(G, S)` for the example graph above will generate 19 socially distant sets, including the following:

[]   [1]   [2]   [1,2]   [2,4]   [1,4]   [1,2,4]

The predicate above can be used for checking if there exists a socially distant set  $S$  for a given graph  $G$  where the size of the set  $S$  is some positive integer  $k$ . For example, the following query does exactly this:

```
?- length(S,  $k$ ), gen_soc_dist(G, S).
```

Alternatively, if  $k$  is small, say 3, one can also issue the following query to check if there is a vertex cover of size 3:

```
?- gen_soc_dist(G, [V1, V2, V3]).
```