CMSC 22100, Spring 2020: Exam

Please complete this exam either writing by hand and pushing a photograph or scan of your responses to your repository, or using a computer strictly as a typewriter and doing the same.

Please note that the word "typewriter" means you shall not use a code editor such as emacs, vim, sublime, atom, vscode, etc., because those editors can be helpful in making syntactic suggestions and/or corrections as you type. The distant exam is meant to simulate the experience of sitting at a desk and writing on paper, so you must use only a simple text editor that has no particular support for code writing.

By submitting the exam you are implicitly certifying that you have scrupulously adhered to the standards of academic honesty. In particular, you are asserting that

- you did not study in advance of taking the exam any time between 7pm CDT on Thursday May 28 and 11:59pm CDT on Friday May 29,

- you wrote your responses in one 80-minute sitting (or more time if by prior special arrangement) in the absence of any study materials ("closed book"), and,

- you did not use a computer to code-edit and/or compile any programs to check their correctness in any way.

The questions follow. Please take care not to write code unless the question has clearly asked for it.

1) Assume a Standard ML definition of the untyped lambda calculus as follows:

```
datatype term
  = Var of string
  | Abs of string * term
  | App of term * term
```

Write a Standard ML function to determine whether a term is stuck or not, assuming call-by-value evaluation semantics. The function should be named isStuck and have type term -> bool.

2) Consider this pair evaluation rule:

```
    t1 -> t1'     t2 -> t2'
------------------------------
(pair t1 t2) -> (pair t1' t2')
```

Assume this is the *only* pair evaluation rule in its evaluation system. While this rule is not wrong, exactly, it has a rather serious practical problem. What is it?

3) Here is a term grammar for a simple language named S for simple.

```
t ::= 0 | 1 | 2 | ...  // natural constants
    | ()               // unit
    | (pair t t)
    | (#1 t)
    | (#2 t)
    | (let x = t in t) // variable bindings
    | x                // variables
```

The types for S are these:

```
tau ::= Nat
      | Unit
      | tau * tau
```

Write the typing judgments (the typing rules) for S.

4) Assume this unusual rule set for evaluation of conditional expressions.

```
                   t3 -> t3'
-----------------------------------------------
if t1 then t2 else t3 -> if t1 then t2 else t3'

                   t2 -> t2'
-----------------------------------------------
if t1 then t2 else v3 -> if t1 then t2' else v3

                   t1 -> t1'
-----------------------------------------------
if t1 then v2 else v3 -> if t1' then v2 else v3

if #t then v2 else v3 -> v2

if #f then v2 else v3 -> v3
```

Why are these rules unusual? Do they present any problems in practice?

5a) In typesafe languages, conditional statements (if statements) are required to have the same type expression in both branches. Why is this important for establishing type safety? Please be specific.

5b) Does this policy mean that certain terms that would never get stuck are nonetheless rejected by the type system?

6) Returning to the language S from question 3: assume type soundness (aka type safety) has been proven about S. Let's add a new term and a new type to the language S, and call the modified language S'.

```
t ::= (everything above) | omega

tau ::= (everything above) | Inf
```

The term omega is not a value. Furthermore there is a typing rule assigning the type Inf to the term omega, and the following evaluation rule:

```
--------------
omega -> omega
```

Would the type soundness result for S still hold for S'? Why or why not?

7) This is the Church encoding of "and":

```
and = λ b . λ c . ((b c) fls)
```

(Note: parentheses are used here for clarity; you may use them too.)

Under this definition of and and the text's definitions of tru and fls, show the full step-by-step CBV evaluation of the following term until it reaches a normal form:

```
and tru fls
```