CS242 Final
Sample Solution
Fall 2022

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, some with multiple parts. You have 180 minutes to work on the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: _____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: _____

Problem	Max points	Points
1	20	
2	25	
3	20	
4	30	
5	15	
TOTAL	110	

1. **π-Calculus** (20 points)

Consider the following version of the π-calculus:

$$\begin{array}{lrcl} \text{Prefixes} & p & ::= & a(x) \\ & & | & \bar{a}x \\ & & | & \text{print}(x) \\ \text{Agents} & P & ::= & 0 \\ & & | & p.\,P \\ & & | & P + P \\ & & | & P \mid P \\ & & | & x = y \Rightarrow P \\ & & | & x \neq y \Rightarrow P \\ & & | & \nu x\, P \\ & & | & !P \end{array}$$

This syntax is identical to Lecture 13, except for the addition of a print(x) prefix that prints the value of x.

(a) Consider the following π-calculus program:

$$\begin{array}{l} \nu c\,(\\ \quad (\bar{c}\,1.\,\bar{c}\,2.\,0) \mid \\ \quad (\bar{c}\,3.\,\bar{c}\,4.\,0) \mid \\ \quad !(c(v).\,\text{print}(v).\,0)) \end{array}$$

Show all possible printed outcomes for this program.

Solution: 4! possibilities: All permutations of 1,2,3,4. This is since no ordering is guaranteed between when we receive from c and print($\cdot$) executes.

We also accepted an alternate solution with 6 total possibilities: 1,2,3,4; 3,4,1,2; 1,3,2,4; 1,3,4,2; 3,1,2,4; 3,1,4,2. This is all possibilities where 1 precedes 2 and 3 precedes 4. This was our original intention, but we realized that we had implemented the last agent incorrectly for this to be true. (Exercise to the reader: what changes would we have needed to make this the correct answer?)

(b) Consider the following π-calculus program:

$$\begin{array}{l} \nu c_1\,\nu c_2\,\nu c_3\,(\\ \quad (\overline{c_1}\,2.\,c_2(\_).\,\overline{c_3}\,1.\,0) \mid \\ \quad (c_1(\_).\,\overline{c_2}\,1.\,\overline{c_3}\,2.\,0) \mid \\ \quad c_3(x).\,\text{print}(x).\,0) \end{array}$$

Show all possible printed outcomes for this program.

Solution: 2 possible outcomes: 1 or 2.

(c) Consider the following π-calculus program:

$$
\begin{aligned}
&\nu c_1\, \nu c_2\, \nu c_3\, (\\
&\quad (\overline{c_1}\, 2.\, c_2(\_).\, \overline{c_3}\, 1.\, 0) \mid \\
&\quad (\overline{c_2}\, 1.\, c_1(\_).\, \overline{c_3}\, 2.\, 0) \mid \\
&\quad c_3(x).\, \mathrm{print}(x).\, 0)
\end{aligned}
$$

Show all possible printed outcomes for this program.

Solution: 1 possible outcome: empty.

Notice that neither of the sends will succeed, since there is no agent receiving from either c_1 or c_2 at the time of the send. In other words, the first two agents are deadlocked.

2. **State** (25 points)

Recall the syntax of the λ-calculus extended with state from lecture 9:

$$e ::= x \mid \lambda x.\, e \mid e\ e \mid i \mid \mathsf{new} \mid !e \mid e := e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$$

In this problem, we extend the syntax with pairs:

$$e ::= \ldots \mid \langle e, e \rangle \mid e.l \mid e.r$$

The following questions ask you to define the semantics of pairs. Your answers should use the "big step" structural operational semantics style of Lecture 9. (In some of the homeworks, particularly HW8, we used a variation called "small step" semantics.)

As a reminder, the structural operational semantics for the λ-calculus with state defines the judgement $E, S \vdash e \to e', S'$, where E is the lexical context that binds the free variables of e to values, S is the initial state, e' is the result of the evaluation of e and S' is the resulting state after evaluation.

(a) Define operational semantics for $e.l$. Your rule should have the following behavior: when the expression e evaluates to the value $\langle v_1, v_2 \rangle$, then $e.l$ should evaluate to v_1.

Solution:

$$\frac{E, S \vdash e \to \langle v_1, v_2 \rangle, S'}{E, S \vdash e.l \to v_1, S'}$$

(b) Define operational semantics for $e.r$. Your rule should have the following behavior: when the expression e evaluates to the value $\langle v_1, v_2 \rangle$, then $e.r$ should evaluate to v_2.

Solution:

$$\frac{E, S \vdash e \to \langle v_1, v_2 \rangle, S'}{E, S \vdash e.r \to v_2, S'}$$

(c) Give an operational semantics rule for the evaluation of $\langle e_1, e_2 \rangle$. Your rule should have the following behavior: when the expression e_1 evaluates to v_1 and the expression e_2 evaluates to v_2. then $\langle e_1, e_2 \rangle$ should evaluate to a pair of v_1 and v_2. There are two possible orders of evaluation; show a rule for each.

Rule 1:

Solution:

$$\frac{E, S \vdash e_1 \to v_1, S' \qquad E, S' \vdash e_2 \to v_2, S''}{E, S \vdash \langle e_1, e_2 \rangle \to \langle v_1, v_2 \rangle, S''}$$

Rule 2:

Solution:

$$\frac{E, S \vdash e_2 \to v_2, S' \qquad E, S' \vdash e_1 \to v_1, S''}{E, S \vdash \langle e_1, e_2 \rangle \to \langle v_1, v_2 \rangle, S''}$$

(d) Does the order of evaluation of pairs matter to the final value? If it does matter, give an example program where the output is different using different orders of evaluation. If it does not matter, explain clearly in one or two sentences why not.

Solution: It does matter. Consider the expression

$$\begin{array}{l} \mathsf{let}\ p = \mathsf{new}\ \mathsf{in} \\ \mathsf{let}\ \_ = (p := 0)\ \mathsf{in} \\ \langle \mathsf{let}\ \_ = (p := 1)\ \mathsf{in}\ 0, !p \rangle. \end{array}$$

Rule 1 will result in $\langle 0, 1 \rangle$, while rule 2 will result in $\langle 0, 0 \rangle$.

3. **Dependent Types** (20 points)

Consider the following λ-calculus with dependent types:

$$e ::= x \mid \lambda x{:}\, t.\, e \mid e\ e \mid \langle e, e \rangle \mid \text{case } e{:}\, (\gamma \vee \delta) \text{ of } \lambda x{:}\, \gamma.\, e, \lambda x{:}\, \delta.\, e$$

This language is missing some constructs (like the deconstructors for pairs); the missing features are not needed to answer this question.

The dependent types t are given by the grammar:

$$t ::= \alpha \mid t \to t \mid t \wedge t \mid t \vee t \mid \forall \alpha.\, t \mid \text{Type}$$

The material for this question is covered in Lecture 14. The essential items from that lecture that you need for this problem are

- The type for a pair $\langle e_1, e_2 \rangle$ is $t_1 \wedge t_2$, where $e_1 : t_1$ and $e_2 : t_2$.
- The value of
$$\text{case } e_0{:}\, (\gamma \vee \delta) \text{ of } \lambda x{:}\, \gamma.\, e_1, \lambda x{:}\, \delta.\, e_2$$
is
  - $(\lambda x{:}\, \gamma.\, e_1)\ e_0$ if $e_0 : \gamma$, and
  - $(\lambda x{:}\, \delta.\, e_2)\ e_0$ if $e_0 : \delta$.
- In the dependently-typed λ-calculus, a polymorphic type $\forall \alpha.\, t$ is the type of functions that take a type argument α and return a value of type t.

For each of the following types, write an expression in the dependently-typed λ-calculus that has that type.

(a) $\alpha \to \beta \to \alpha$
Solution: $\lambda x{:}\, \alpha.\, \lambda y{:}\, \beta.\, x$

(b) $\alpha \to \beta \to \alpha \wedge \beta$
Solution: $\lambda x{:}\, \alpha.\, \lambda y{:}\, \beta.\, \langle x, y \rangle$

(c) $\forall\gamma.\,\gamma \to \gamma$

Solution: $\lambda\gamma\colon \text{Type}.\,\lambda x\colon \gamma.\,x$

(d) $(\alpha \to \gamma) \to (\beta \to \gamma) \to (\alpha \vee \beta) \to \gamma$

Solution:

$$\begin{array}{l}\lambda f\colon \alpha \to \gamma.\,\lambda g\colon \beta \to \gamma.\,\lambda x\colon \alpha \vee \beta.\\ \text{case } x\colon (\alpha \vee \beta) \text{ of}\\ \quad \lambda y\colon \alpha.\,f\ y,\\ \quad \lambda y\colon \beta.\,g\ y\end{array}$$

4. **Continuations** (30 points)

(a) **CPS transformation**

Consider the λ-calculus extended with Boolean values and conditionals:

$$e ::= x \mid e\ e \mid \lambda x.\, e \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$$

In this language, `true` and `false` are primitive values rather than Church encodings. Given an environment E binding free variables in an expression to values, the operational semantics for conditionals are:

$$\frac{E \vdash e_1 \to \texttt{true} \quad E \vdash e_2 \to v}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to v}\ \text{[If-True]} \qquad \frac{E \vdash e_1 \to \texttt{false} \quad E \vdash e_3 \to v}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to v}\ \text{[If-False]}$$

Recall the CPS transformation for λ-calculus covered in Lecture 11:

$$\begin{aligned} C(\lambda x.\, e, k) &= k\ (\lambda k'.\, \lambda x.\, C(e, k')), \\ C(e\ e', k) &= C(e, \lambda f.\, C(e', \lambda v.\, f\ k\ v)), \\ C(x, k) &= k\ x. \end{aligned}$$

Write the CPS transformation rules for the new expressions. To avoid wasted work, make sure that your implementation of `if` expressions only evaluates one branch – either e_2 or e_3.

$$C(\texttt{true}, k) =$$

$$C(\texttt{false}, k) =$$

$$C(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, k) =$$

Solution:

$$\begin{aligned} C(\texttt{true}, k) &= k\ \texttt{true}, \\ C(\texttt{false}, k) &= k\ \texttt{false}, \\ C(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, k) &= C(e_1, \lambda v.\, \texttt{if}\ v\ \texttt{then}\ C(e_2, k)\ \texttt{else}\ C(e_3, k)). \end{aligned}$$

(b) **While loop**

We want to implement a `while` loop in Racket using using `call/cc`. Given two functions `cond` and `body`, the function call (`while cond body`) should repeatedly evaluate `cond` and, if the result of `cond` is `true`, then evaluate `body`. The loop terminates if `cond` evaluates to `false`. The return value of `while` can be any value.

As an example, running the first loop below prints "`test`" exactly four times. The second loop produces no output, since the condition is always `false`. Equivalent code in Python is included for reference.

```
(define i 0)                        ; i = 0
(while (lambda () (< i 4))          ; while i < 4:
       (lambda ()                   ;
         (printf "test\n")          ;   print("test")
         (set! i (+ i 1))))         ;   i = i + 1

(while (lambda () #f)               ; while False:
       (lambda ()                   ;
         (printf "unreached\n")))   ;   print("unreached")
```

For each implementation below, determine if it correctly implements `while`. If the implementation is correct, explain in no more than three sentences how it works. If the implementation is incorrect, give any example `while` loop of no more than five lines that doesn't work correctly and explain in no more than two sentences what goes wrong.

i.
```
(call/cc (lambda (k)
  (body)
  (if (cond)
    (k)
    #f)))
```

Solution: Incorrect. Second example above. "`unreached`" is printed once.

ii.
```
(define cont 0)
(if (cond)
  (begin
    (call/cc (lambda (k) (set! cont k)))
    (body)
    (cont))
  #f)
```

Solution: Incorrect. In the first example above, "`test`" is printed infinitely many times.

Here is an implementation that actually works properly:

```
(define (while cond body)
  (define cont 0)
  (call/cc (lambda (k) (set! cont k)))

  (if (cond)
    (begin
      (body)
      (cont))
    #f))
```

5. **Type State** (15 points)

In the game tic-tac-toe, two players take turns placing symbols on a 3×3 board. One player always places the character `X`, while the other always places the character `O`. Assume that `X` always goes first at the start of the game.

A player wins if they are able to place three of their respective characters in consecutive locations either vertically, horizontally, or diagonally. The game can also end in a draw if all locations in the grid are filled without a winner.

Below are examples of game states ending in an `X` win, an `O` win, and a draw, respectively.

X	O	O
	X	
		X

O		X
O	X	X
O		

X	X	O
O	X	X
X	O	O

Suppose that you are implementing an API to play a game of tic-tac-toe with a friend. Your friend proposes the following design, representing the game through a `TicTacToeBoard` object:

- `CreateTicTacToeBoard() → TicTacToeBoard`: Constructor to create a new game board.
- `placeX(row: uint8, col: uint8) → bool`: Place an `X` at the given location and return `true` if the game is over, or `false` otherwise.
- `placeO(row: uint8, col: uint8) → bool`: Place an `O` at the given location and return `true` if the game is over, or `false` otherwise.

  To simplify the problem, we ignore the differences between `X` winning, `O` winning, and a draw.
- `clearBoard()`: Clear the board and start a new game.

However, you have some concerns about this interface. A dishonest opponent might try to call `placeX` or `placeO` out of turn to make multiple moves in a row. In addition, they may decide to call `clearBoard` out of spite to end the game early.

Fortunately, you've taken CS 242 and learned how to use type state (Lecture 10) to enforce the rules of the game. **Give a type state machine diagram that satisfies the following requirements:**

- The transitions between states in your design are only `CreateTicTacToeBoard`, `clearBoard`, `placeX[true]` (which means `placeX` returned `true`), `placeX[false]`, `placeO[true]`, and `placeO[false]`.
- `placeX[...]` should only be possible if it is player X's turn, and `placeO[...]` should only be possible if it is player O's turn.
- Clients should only be able to clear the board once the game is over.

Note that the problem is only to give the type state machine diagram, not to give code that implements tic-tac-toe.

(your problem 5 answer goes here)

Solution: