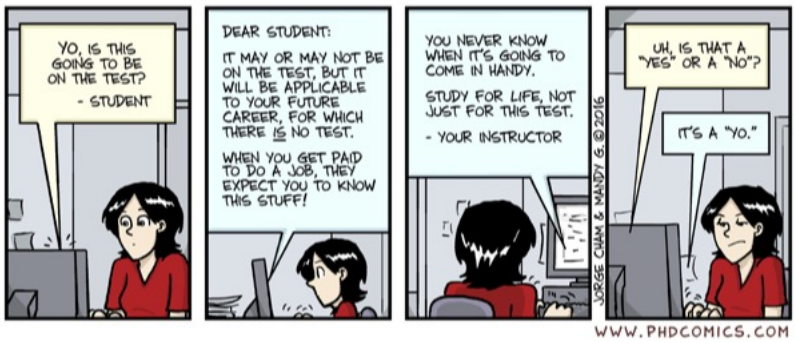
CS152: Abstractions and Paradigms for Programming

End-Sem Exam (2.5 hours; 45 marks)

November 17^{th}, 2023

Instructions:

- Write neat, clear and crisp answers.
- In case you make an assumption, write it down before the corresponding answer.



A [14]: DON'T MEMORISE; CREATE MEMORIES

A1 [4] Match the columns based on memories from CS152 classes:

(a) Haskell	(i) Snape
(b) Prolog	(ii) Sugar
(c) Python	(iii) Spectacles
(d) Java	(iv) Duck

A2 [2] "Short programs may not be sweet." Comment based on your experience with the logic paradigm.

A3 [5] Determine the outputs of the programs P1, P2, P3, P4 and P5 that use continuations:

```
; P1
(+ 4 (call/cc (lambda (cont) 13)))

; P2
(+ 4 (call/cc (lambda (cont) (cont 10) 13)))

; P3
(define foo #f)
(+ 4 (call/cc (lambda (cont) (set! foo cont) 13)))

; P4 (in sequence with P3)
(foo 100)

; P5 (in sequence with P4)
(foo 416)
```

A4 [3] For the following fibonacci function:

$$F = \lambda f.\lambda n.\ if\ (<\ n\ 2)\ n\ (+\ (f\ (-\ n\ 1))\ (f\ (-\ n\ 2)))$$

show that $((Y\ F)\ n)$ correctly computes fibonacci(n), where Y is the Y-combinator. Do not skip any step.

B [19]: THINK THROUGH THY THOUGHTS

B1 [2] The following on a Scheme prompt would tell that `foo` is a procedure:

```
> (define (foo x y) (+ x y))
> foo
```

whereas the following on a Haskell prompt would throw an error:

```
> foo x y = x + y
> foo
```

Explain why.

B2 [6] Spiderman says pure functional programming cannot achieve anything useful. Pikachu disagrees and claims to have written several "effectful" programs in Haskell. The world of elves has a special tool called "monad" that can reconcile the world of superheros created by CS152.

Can you make this discussion fruitful, by explaining (a) what is functional purity and why is it good; (b) what is a monad; and (c) how does Haskell support writing effectful programs?

B3 [6] (a) Consider an anagram dictionary that contains entries like the following:

```
...
eginor: ignore,region
eginrr: ringer
eginrs: resign,signer,singer
...
```

Thus, the letters of the anagrams are sorted and the results are stored in dictionary order. Associated with each anagram are the English words with the same letters. Describe how would you go about designing a Haskell function

```
anagrams :: Int -> [Word] -> String
```

so that `anagrams n` takes a list of English words in alphabetical order, extracts just n-letter words and produces a string that, when displayed, gives a list of the anagram entries (like above) for the n-letter words. For example, the above could be part of the output of `anagram 6` (for 6-letter words) applied to a list that contains all those words as shown above apart from few more words of varying lengths. You do not need to define individual functions; just give suitable names and types for each function that you would use, along with what is that function supposed to do, and finally the composition that can be performed to get `anagrams n`. [5]

(b) Decipher the following anagram: [1]

¯_ *HINKA RAU BIAHK IFR EEGLINM* _/¯

B4 [5] Consider the following Prolog facts and rules:

```
bird(sparrow).
bird(penguin).
fly(penguin) :- !, fail.
fly(X) :- bird(X).
```

(a) Explain the outputs of the following queries: [2]

(i) `fly(penguin).`

(ii) `fly(X).`

(b) Change the code to correctly infer that though penguins cannot fly, if someone asks for a list of birds that can fly they indeed get a sparrow. [2]

(c) What is the general problem with declarative systems like Prolog that necessitates the above change? [1]

C [12]: DO YOUR OWN LANGUAGES

C1 [4] Write a purely functional Scheme program that is not straightforward to write in Haskell. What is the "nice" property about Haskell that creates this "issue"? In which scenarios would you prefer and not prefer this property in the programming language you are working with (or designing!), and why?

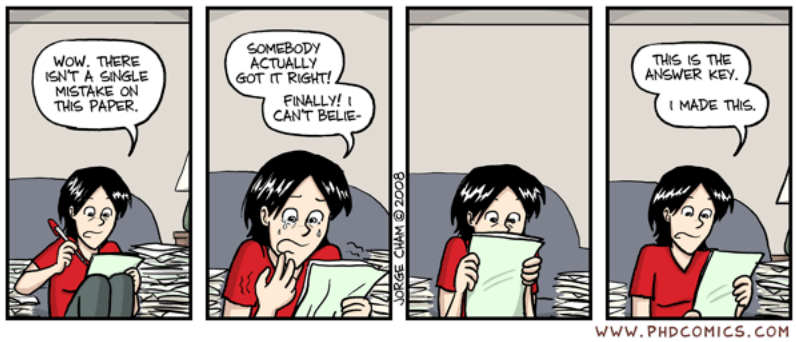
C2 [3] What simplifications could be done to the Scheme interpreter if Scheme did not have assignment statements? Write your answer in terms of the changes in the `eval` and `apply` procedures.

C3 [4] Recall the discussion from PA4 about logic being "flow insensitive". We had concluded that as a result of this property, the following testcase would *imprecisely* return `true`:

```
alloc(a, o1).
alloc(b, o2).
copy(a, b).
invoke(m, [a]).
? escapesthrough(o1, m).
```

How would you modify PA4 such that, while being within Prolog's limitation, you would still be able to compute "flow sensitive" information?

C4[1] Complete the sentence: "A BTech without CS152 be like .....".



——— *HO GAYA SAMAAPT!* ———