CMSC330 Spring 2024 Lecture Quizzes for Cliff

Ash Dorsey
ash@ash.lgbt

Contents

Lecture Quiz 1/25

1. How many 24-hour extension tokens do you get at the start of the semester?
   - ○ 6
   - ○ 3
   - ○ 4
   - ○ 7
2. Within how many days after an assignment is returned can you submit a regrade request?
   - ○ 7
   - ○ 5
   - ○ 10
   - ○ 3
   - ○ until the end of time
   - ○ until the end of the semester
3. When is the first quiz?
   - ○ 7th February, 2024
   - ○ 9th February, 2024
   - ○ 10th February, 2024
   - ○ 8th February, 2024
4. How many tokens can you use per project?
   - ○ 3
   - ○ 2
   - ○ 1
   - ○ As many as you have
5. If you have to take a makeup quiz or exam what is the policy?
   - ○ Hope the professor's notice you were out and email you
   - ○ Post on Piazza about how you need a makeup
   - ○ Email all the TA's and Professors
   - ○ Fill out the form on the syllabus, provide documentation, and make sure you receive confirmation you were signed up for a makeup

Answers

Q1: 3
Q2: 3
Q3: 9th February, 2024
Q4: 1
Q5: Fill out the form on the syllabus, provide documentation, and make sure you receive confirmation you were signed up for a makeup

Lecture Quiz 2/1

1. Ocaml is a compiled language.
   - ○ True
   - ○ False

2. What type of typing system does Ocaml follow?
   - ○ Static and latent (implicit).
   - ○ Static and manifest (explicit).
   - ○ Dynamic and latent (implicit).
   - ○ Dynamic and manifest (explicit).

3. What is true about the following code segments?

   3.1.
   ```
   let x = 3.0;;
   let z = x +. 4;;
   let x = 1;;
   x + z;;
   ```
   - ○ This code has 1 variable called x, and it is bound to 1 and the last line compiles because Ocaml supports addition between integers and floats.
   - ○ This code has 1 variable called x, and it does not compile because x is an int and z is a float in the last line.
   - ○ This code has 2 variables called x. The x = 1 shadows the x = 3.0 binding and the last line evaluates to 8
   - ○ This code has 2 variables called x. The x = 1 shadows the x = 3.0 and it does not compile because x is a an int and z is a float in the last line.
   - ○ None of the above

   3.2.
   ```
   let x = 3.0;;
   let x = 4;;
   let z = 1;;
   x + z;;
   ```
   - ○ This code has 1 variable called x, and it is bound to 1, and the last line evaluates to 3.0.
   - ○ This code has 1 variable called x, and it does not compile because both x and z are integers.
   - ○ This code has 2 variables called x, and since the x = 3.0 shadows the x = 4 binding the last line evaluates to 5.
   - ○ This code has 1 variable called x, and the last line evaluates to 5.
   - ○ None of the above

4. Answer the following questions using this code:

   ```
   [1]::[2]
   ```

   4.1. Does the code output [[1];[2]]?

   A) No, because we cannot cons (the :: operator) two variables of the same type together
   B) Yes
   C) No, because in order to get the desired output [2] should have been [[2]]
   - ○ A, C
   - ○ B
   - ○ A

- ◯ C
- ◯ All of the above
- ◯ None of the above

4.2. If the above code doesn't compile then, which of the following codes will help resolve the issue?

A) `[[1]]@[[2]]`
B) `[1]::[[2]]`
C) `[2]::[[1]]`
D) `1::[2]`

- ◯ A, B
- ◯ A, B, C
- ◯ B, C
- ◯ C
- ◯ D
- ◯ B, D
- ◯ All of the above
- ◯ The code compiles with no issues

5. What is the type of the following expressions:

5.1. `let func a b c = a + b * c`

- ◯ `int -> int -> int -> int`
- ◯ `'a -> 'a -> 'a -> int`
- ◯ `'a -> 'a -> 'a -> 'a`
- ◯ `int -> int -> int -> 'a`
- ◯ None of the above

5.2.
```
if (2 > 3) then
   (match 3 with 4 -> "a" | 5 -> "b" | _ -> "c")
else
   (if false then "x" else "y")
```

- ◯ `string`
- ◯ `int`
- ◯ `int -> int -> string`
- ◯ `'a`
- ◯ `int -> 'a`
- ◯ None of the above

5.3. What is the type for the following OCaml expression?

```
let g f b =
   if (f 1 2) = 3 && b = 7.5 then 7 else 5
```

- ◯ `int -> int -> int -> int -> float`
- ◯ `(int -> int -> int) -> float -> int`
- ◯ `int -> int -> int -> float -> int`
- ◯ None of the above

6. Consider the following code as it changes:
   6.1. Initial code:

   ```
   let f a = match a with (a, b) -> (a, a)
   ```

   What is the type of this code?

   ○ `'a * 'b -> 'a * 'a`

   ○ `'b * 'b -> 'a * 'a`

   ○ `'a * 'a -> 'b * 'b`

   ○ `'a * 'b -> 'b * 'a`

   ○ None of the above

   6.2. Changed code:

   ```
   let f a = match a with (a, b) -> if b then (a, a) else (b, b)
   ```

   ○ `bool * bool -> bool * bool`

   ○ `'a * 'b -> 'a * 'a`

   ○ `'a * bool -> bool * bool`

   ○ `bool * 'b -> bool * bool`

   ○ None of the above

   6.3. What causes the change, if any, in the types?

   ○ There is no change in types.

   ○ The `if` expression checks if `b` is true which means that `b` has to be a `bool`. Since the function also returns `(b, b)` the return type has to be of type `bool * bool`

   ○ The `if` expression checks if `b` is true which means that `b` has to be a `bool`. The return type stays the same

   ○ The `if` expression checks if `b` is true which means that `b` has to be a `bool`. We cannot assume the type of `'a` from here since it is never operated on.

7. Which of the following functions takes in a non-empty list and returns the head?

   ○ `let f lst = match lst with h::t -> h`

   ○ `let f lst = fst lst`

   ○ `let f lst = match lst with h::t -> t`

   ○ `let f lst = match lst with h -> h | h::t -> h`

8. Your coworker wrote the following function on OCaml:

   ```
   let mult_twice x y = x * x * y * y;;
   ```

   You try to call it like so:

   ```
   let result = mult_twice (2, 3);;
   ```

   Does this compile?

   ○ No, both the parentheses and comma must be removed, otherwise Ocaml will think `(2, 3)` is a tuple.

   ○ No, only the comma must be removed, otherwise Ocaml will think `(2, 3)` is a tuple.

   ○ No, there must be no space between `mult_twice` and the parentheses.

   ○ Yes

9. Which one of the following expressions evaluates to `false`?
   - ◯ `let x = 4 in if x = 4 then if false then false else true else false`
   - ◯ `let (a, b) = (1, 2) in if a = b then false else if true then false else true`
   - ◯ `if if true then false else true then false else if true then true else false`
   - ◯ `if true then true else if true then false else false`

10. The `let expression` is an extension of the `let binding` which we talked about. An example of the let expression is `let x = 4 in x + 5`.

    Answer the following questions about `let expressions`. You can play around with the following code or for more information, you can read section 1.3.3 of Cliff's notes.

    10.1. `let x = 4 in x + 5` is an expression. What does that mean about the code segment?

    A) `let x = 4 in x + 5` evaluates to a value and has type
    B) `let x = 4 in x + 5` could be used wherever we expect an expression (for example: `2 + (let x = 4 in x + 5)`) is valid)
    C) `let x = 4 in x + 5` must be a value

    - ◯ A
    - ◯ B
    - ◯ C
    - ◯ A,B
    - ◯ A,C
    - ◯ B,C
    - ◯ A,B,C
    - ◯ None of the above

    10.2. What is true about the following expression:

    ```
    let x = 3 in let x = 4 in x + 7
    ```

    - ◯ It evaluates to 11 because `x = 4` shadows `x = 3`
    - ◯ It evaluates to 10 because `x + 7` considers the first binding over more recent ones
    - ◯ None of the above

    10.3. Does the following code compile?

    ```
    let x = 3 in x + 6;;
    x + 7
    ```

    - ◯ Yes and the last line evaluates to 10
    - ◯ No because the scope of `x` ends when the `let expression` ends
    - ◯ None of the above

    10.4. Does the following code compile?

    ```
    let x = 6;;
    let x = 3 in x + 6;;
    x + 7
    ```

    - ◯ Yes, and `let x = 3 in x + 6` evaluates to 9, and `x + 7` evaluates to 13
    - ◯ Yes, and `let x = 3 in x + 6` evaluates to 9, and `x + 7` evaluates to 10
    - ◯ No because the `x` becomes unbound after the `let expression`
    - ◯ None of the above

10.5. The following has a type error:

```
if (let x = 3 in x > 5) then
  (let x = 5 in let y = 6 in x + y)
else
  (let x = (match [true;false] with
     [] -> -1
    |x::xs -> 2) in
   -x)
```

◯ Yes

◯ No

10.6. The following code evaluates to a `bool` type

```
let f x y = x + y in
let g x y = x * y in
(f 2 1) > (g 1 2)
```

◯ Yes, and the value is `false`

◯ Yes, and the value is `true`

◯ No, but it does compile

◯ No, because it does not compile

Answers

Q1: True
Q2: Choice 1 of 4:Static and latent (implicit).
Q3.1: None of the above
Q3.2: This code has 2 variables called x, and since the x = 3.0 shadows the x = 4 binding the last line evaluates to 5.
Q4.1: A, C
Q4.2: A, B
Q5.1: `int -> int -> int -> int`
Q5.2: `string`
Q5.3: `(int -> int -> int) -> float -> int`
Q6.1: `'a * 'b -> 'a * 'a`
Q6.2: `bool * bool -> bool * bool`
Q6.3: The `if` expression checks if `b` is true which means that `b` has to be a `bool`. Since the function also returns `(b, b)` the return type has to be of type `bool * bool`
Q7: `let f lst = match lst with h::t -> h`
Q8: No, both the parentheses and comma must be removed, otherwise Ocaml will think `(2, 3)` is a tuple.
Q9: `let (a, b) = (1, 2) in if a = b then false else if true then false else true`
Q10.1: A,B
Q10.2: It evaluates to 11 because `x = 4` shadows `x = 3`
Q10.3: No because the scope of `x` ends when the `let expression` ends
Q10.4: Yes, and `let x = 3 in x + 6` evaluates to 9, and `x + 7` evaluates to 13
Q10.5: No
Q10.6: Yes, and the value is `true`

Lecture Quiz 2/8

1. The next few questions pertain to `fold`.

   Here is the fold function from class for reference.

```
let rec fold f a l = match l with
|[ ] -> a
|h::t -> fold f (f a h) t
```

   1.1. Which of the following fold implementations returns the concatenated form of a list of strings?

   - ◯ `let joined = fold_left (fun a x -> a , x) "" lst`
   - ◯ `let joined = fold_left (fun a x -> a ^ x) "" lst`
   - ◯ `let joined = fold_left (fun a x -> a + x) "" lst`
   - ◯ `let joined = fold_left (fun a x -> a::x) "" lst`
   - ◯ `let joined = fold_right (fun a x -> a + x) "" lst`
   - ◯ `led joined = fold_right (fun a x -> a ^ x) "" lst`

2.
```
let y = match (2, true) with
   | (1, _)       -> "one"
   | (2, false)   -> "two"
   | (_, _)       -> "three"
   | (_, true)    -> "four"
;;
```

   What is the binding of `y`?

   - ◯ "three"
   - ◯ "one"
   - ◯ "two"
   - ◯ "four

3. What is the type of this expression?

```
let rec map f l = match l with
   [] -> []
  |x::xs -> (f x)::(map f xs)
in
map (fun x y -> x + y) [1;2;3];
```

   - ◯ `int -> int -> 'a list`
   - ◯ `int -> int list`
   - ◯ `(int -> int) list`
   - ◯ `('a -> int) list`

4. The following questions pertain to `foldr` whose definition is given below:

```
let rec foldr f l a = match l with
[] -> a
|h::t-> f h (foldr f t a)
```

   4.1. Which of the following is the type of the fold right function?

   - ◯ `('b -> 'b -> 'a) -> 'b list -> 'a -> 'a`
   - ◯ `('a -> 'b -> 'a) -> 'a list -> 'b -> 'a`
   - ◯ `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

○ `('b -> 'a -> 'a) -> 'b list -> 'a -> 'a`

4.2. Given the function:

```
let sum_powers a b =
  match a with
  (c, d) -> (c + 1, d@[b * c])
```

Does this function work properly (return the same result) for both `fold` and `foldr`, why or why not?

○ Yes, the indexing works is uniform no matter what direction you iterate through a list
○ No, the types are wrong
○ No, the indexing doesn't work backwards for `foldr`
○ Yes, it works because `fold` and `foldr` are the same and indexing works both ways.

5. What is the output of the following OCaml code?

```
let lst = [1;2;3;4;5;6;7;8] in
fold (fun acc x -> if x mod 2 = 0 then x::acc else acc) [] lst
```

○ `[1;2;3;4]`
○ `[2;4;6;8]`
○ `[2;5;8;3]`
○ `[8;6;4;2]`

6. Below, we have an implementation of `div3` which takes in a list of integers and returns a list of bools where corresponding indices indicate that a number is divisible by 3. For example:

`input: [1;2;3;4;5;6]` `output:[false;false;true;false;false;true]`

Implementation:

```
let div3 lst =
  map (fun x -> if x mod 3 = 0 then true else false) lst
```

This implementation uses the "map" HOF discussed in lecture. Which of the following "fold" implementations is equivalent?

○ `fold (fun a x -> if x mod 3 = 0 then true::a else false::a) [] lst`
○ `fold (fun a x -> if x mod 3 = 0 then a @ [true] else a @ [false]) [] lst`
○ `fold (fun a x -> if x mod 3 = 0 then true else false) [] lst`
○ `fold (fun a x -> if x mod 3 = 0 then a @ [false] else a @ [true]) [] lst`

7. Which of the following are true about `fold` and `foldr`? (Select all that apply)

7.1. both `fold` and `foldr`'s order for parameters is function, initial accumulator, and list.
7.2. `fold` is tail recursive while `foldr` is not
7.3. for the (-) function (given a list of integers), `fold` and `foldr` would produce different results
7.4. for the (+) function (given a list of integers), `fold` and `foldr` would produce different results
7.5. the function passed to `fold` takes in accumulator then item, and the function passed to `foldr` takes in item then accumulator

○ 1, 2, 4
○ All of the choices

- ◯ 2, 4
- ◯ 3, 4, 5
- ◯ 1, 3, 5
- ◯ None of the choices
- ◯ 2, 3, 4

8. Which of the following Ocaml expressions matches the following type: `('a -> 'b) -> ('b -> 'c) -> 'a -> 'c`?
   - ◯ `fun f g x -> x (g f)`
   - ◯ `fun f g x -> x (f + g)`
   - ◯ `fun f g x -> f (g x)`
   - ◯ `fun f g x -> g (f x)`

9. Which of the following functions is a tail recursive function?

```
(A)
let rec trib n a b c =
  if n = 0 then c else trib (n - 1) b c (a + b + c)

(B)
let rec trib n =
  if n = 0 then trib (n - 1) + trib (n - 2) + trib (n - 3)
  else 0

(C)
let rec trib n a b c =
  if n = 0 then
    trib (n - 1) b c (a + b + c) +
    trib (n - 2) b c (a + b + c) +
    trib (n - 3) b c (a + b + c)
  else
    c
```

   - ◯ A
   - ◯ All of the above
   - ◯ B
   - ◯ None of the above
   - ◯ C

Answers

Q1.1: `let joined = fold_left (fun a x -> a ^ x) "" lst`
Q2: `"three"`
Q3: `(int -> int) list`
Q4.1: `('b -> 'a -> 'a) -> 'b list -> 'a -> 'a`
Q4.2: No, the types are wrong
Q5: `[8;6;4;2]`
Q6: `fold (fun a x -> if x mod 3 = 0 then a @ [true] else a @ [false]) [] lst`
Q7: None of the choices
Q8: `fun f g x -> g (f x)`
Q9: A

Lecture Quiz 2/16

1. Say we define a variant type called tree, where subtype Node is in the form `(left subtree, integer value, right subtree)`:

```
type tree =
  | Node of tree * int * tree
  | Leaf
```

For instance, a tree like this one:

```
         ___4___
     ___6___    Lf
  _5_      _7_
Lf   Lf  Lf   Lf
```

Would be written as:

```
Node(
  Node(
    Node(
      Leaf, 5, Leaf
    ),
    6,
    Node(
      Leaf, 7, Leaf
    )
  ),
  4,
  Leaf
)
```

1.1. Which of the following is the proper implementation of an inorder traversal of this tree that returns a list of the elements?

```
(A) let rec f t = match t with
      |Leaf -> 0
      |Node(l,v,r) -> (f l) + v + (f r)
(B) let rec f t = match t with
      |Leaf -> []
      |Node(l,v,r) -> (f l) @ [v] @ (f r)
(C) let rec f t = match t with
      |Leaf -> []
      |Node(l,v,r) -> (f l) :: [v] :: (f r)
(D) let rec f t = match t with
      |Leaf -> []
      |Node(l,v,r) -> (f l) @ [v] :: (f r)
```

- ◯ D
- ◯ A
- ◯ None of the given functions
- ◯ C
- ◯ B

1.2. Which of the following functions properly adds the values of every node in the tree?

```
(A) let rec f t = match t with
      |Leaf -> 0
      |Node(l,v,r) -> v + (f l) + (f r)
(B) let rec f t = match t with
      |Leaf -> 0
      |Node(l,v,r) -> (f l) + (f r)
(C) let rec f t = match t with
      |Leaf -> 0 + (f t)
      |Node(l,v,r) -> v + (f l r)
(D) let rec f = match t with
      |Leaf -> 0
      |Node(l,v,r) -> v + l + r
```

- ◯ A
- ◯ D
- ◯ C
- ◯ B

1.3. Which of the following would have the same behavior as `map` but for trees? In other words, given a function and a tree, which of these will successfully return a tree where all node values have been updated using the function?

```
A) let rec tree_map f t = match t with
    |[] -> []
    |x::xs -> (f x)::(tree_map f xs)
B) let rec tree_map f t = match t with
    |Leaf -> []
    |Node(l,v,r) -> f (tree_map f l) (v) (tree_map f r)
C) let rec tree_map f t = match t with
    |Leaf -> Leaf
    |Node(l,v,r) -> (tree_map f l),(f v),(tree_map f r)
D) let rec tree_map f t = match t with
    |Leaf -> Leaf
    |Node(l,v,r) -> Node(tree_map f l, f v, tree_map f r)
```

- ◯ B
- ◯ D
- ◯ Multiple of the above
- ◯ None of the above
- ◯ A
- ◯ C

2. Consider the modified llist type from lecture:

```
type 'a llist = Nil|Cons of ('a * 'a llist)
```

Use this definition for the following questions

2.1. Consider how we derived `fold` for lists. We now want to derive a `llist_add` function that adds up all the values in a `llist`. You can assume `f` is a function that adds up the `'a` type for each `llist` type and that the list has at least 1 element:

```
A) let rec llist_add llst f = match llst with
     |[] -> 0
```

```
        |x::xs -> x + llist_add xs f
B) let rec llist_add llst f = match llst with
        |Nil -> 0
        |Cons(value, rest) -> value + (llist_add rest f)
C) let rec llist_add llst f = match llst with
        |Cons(x, Nil) -> x
        |Cons(value, rest) -> f value llist_add rest f
        |_ -> failwith "This should never happen"
D) let rec llist_add llst f = match llst with
        |Cons(x, Nil) -> x
        |Cons(value, rest) -> f value (llist_add rest f)
        |_ -> failwith "This should never happen"
```

- ◯ Multiple of the above
- ◯ A
- ◯ D
- ◯ B
- ◯ None of the above
- ◯ C

2.2. We cannot use `List.fold_left` nor `List.fold_right` to add `llist`'s up. Why?

- ◯ The type of the fold functions expect a list to iterate through, not a variant
- ◯ This is a false statement. We can use `fold_left` and `fold_right`
- ◯ this is a false statement. We can not use `fold_left`, but we can use `fold_right`
- ◯ None of the above answer choices correctly answer the question
- ◯ This is a false statement. We can use `fold_left` but not `fold_right`

2.3. Let's write a custom fold function that has the same functionality of `fold_left`.

Which of the following is valid?

```
A) let rec llist_fold f a llst = match llst with
        |[] -> a
        |x::xs -> llist_fold f (f a x) xs
B) let rec llist_fold f a llst = match llst with
        |Nil -> a
        |Cons(x,r) -> f x (llist_fold f a r)
C) let rec llist_fold f a llst = match llst with
        |Nil -> a
        |Cons(x,r) -> llist_fold f (f a x) r
```

- ◯ Multiple of the above
- ◯ A
- ◯ B
- ◯ C
- ◯ None of the above

2.4. Let's use our custom fold function to add up a `int llist`. Which of the following are valid?

```
A) let add_llist llst = llist_fold (fun a x -> a + x) 0 llst
B) let add_llist llst = llist_fold (fun x a -> a + x) 0 llst
C) let add_llist llst = llist_fold (+) 0 llst
```

- ○ A,C
- ○ A
- ○ B
- ○ C
- ○ B,C
- ○ None of the answer choices
- ○ A,B
- ○ A,B,C

3. Observe the following variant type:

```
type bruh =
  | Slayp of int * bruh
  | Nfdl
```

3.1. Which of the following evaluates to a `bruh` type?

- ○ `Slayp(3, Slayp(Nfdl, 4))`
- ○ `Slayp(2, Slayp(3, Slayp(4, Slayp(5, Nfdl))))`
- ○ `Slayp(Nfdl(Slayp(1, Nfdl)),Nfdl)`
- ○ `Slayp(Nfdl, Nfdl)`

3.2. Now, observe the following function, designed to sum all the elements in a bruh type:

```
let rec sum b =
  match b with
    | Nfdl -> 0
    | Slayp (number, b) -> (sum number) +. (sum b)
```

What is wrong with this code?

1. the match statements return different types
2. we don't have enough match statements
3. `sum` is called on an integer instead of a `bruh` type
4. `sum` is called on a `bruh` type instead of an integer
5. while matching, we should have `Nfdl (number, b)`, but `Slayp` by itself

- ○ 1
- ○ 1, 2, 3
- ○ 5
- ○ None of the other option choices
- ○ 1, 3
- ○ 2
- ○ 2, 4
- ○ 3
- ○ 4
- ○ 2, 4, 5

4. Answer the following questions using the types declared below:

```
type student = {
   name: string;
   graduation_year: int;
   major: string
};;

type course = {
   course_code: string;
   num_students: int;
   students: student list
};;
```

Assume that `union` and `intersection` function operate in by taking in 2 lists and returns a list combined in the same way set union and set intersection work.

4.1. Which function that recursively iterates through a list of courses and returns a unique list of all the students? You can assume that each student list within each `course` is unique to begin with.

```
(A) let rec get_student_names lst = match lst with
    |[] -> []
    |h::t -> intersection h (get_student_names t)
(B) let rec get_student_names lst = match lst with
    |[] -> []
    |h::t -> intersection h.students (get_student_names t)
(C) let rec get_student_names lst = match lst with
    |[] -> []
    |h::t -> union h.students (get_student_names t)
(D) let rec get_student_names lst = match lst with
    |[] -> []
    |h::t -> union h (get_student_names t)
```

- ◯ D
- ◯ B
- ◯ None of the given functions
- ◯ A
- ◯ C

4.2. Write a function that achieves the same result, but using `fold`. The definition of fold is given below:

```
let rec fold f acc lst = match lst with
   |[] -> acc
   |x::xs -> fold f (f acc x) xs
```

```
(A) let get_student_names lst =
    fold (fun acc x -> intersection acc x.students) [] lst
(B) let get_student_names lst =
    fold (fun acc x -> intersection acc x) [] lst
(C) let get_student_names lst =
    fold (fun acc x -> union acc x) [] lst
(D) let get_student_names lst =
    fold (fun acc x -> union acc x.students) [] lst
```

- ◯ None of the given functions
- ◯ A

◯ B
◯ C
◯ D

5. Consider the following code

```
let y = ref 1;;
let f x y = x + y;;
let w = f (y:=2;!y) (!y);;
```

5.1. What is the value of w if f's arguements are evaluated left to right?
   ◯ 2
   ◯ None of the answer choices
   ◯ 4
   ◯ 3

5.2. What is the value of w if f's arguements are evaluated right to left?
   ◯ 2
   ◯ None of the answer choices
   ◯ 4
   ◯ 3

6. The following declaration of the variable will define the fields of our record to be mutable

```
type home_coordinates = {x:int; y:int; c:string};;
```

◯ True
◯ False

7. In the following section of code we are passing a reference of variable x to the new variable y

```
let x = 7;
let y = x;
```

◯ False
◯ True

8. What is the output we get on running the following code?

```
let x = ref 5 in
let y = ref (fun z -> z + !x) in
let z = ref (fun w -> !y w + w) in
let _ = x := 10 in
let _ = y := (fun z -> z * 2) in
let _ = z := (fun w -> !y w * 3) in
let result = !z 3 in
result
```

◯ 25
◯ 18
◯ 14
◯ 15

9. Which of the following is NOT a valid variant type declaration?

- ○ `type two = Fish of int, int`
- ○ `type one = Fish of (bool -> int)`
- ○ `type blue = Fishie | Fisher of int | Fishee`
- ○ None of the above
- ○ `type red = int list`

10. Answer the question given the following record definition and instance:

```
type animal = {legs: int; eyes: int; name: string; spine: bool;
fur: bool; lastname: string}
let spider = {legs = 8; eyes = 8; name = "Joe"; spine = false;
fur = true; lastname = "Hills"}
```

Which of the following successfully return `"Joe"`, the name of our spider?

```
A) let {name} = spider in name

B) let {spine=_; name = x} = spider in name

C) match spider with {lastname = "Biden"} -> lastname |
{name = _} -> name

D) match spider with {lastname = "Biden"} -> spider.lastname |
{name = _} -> spider.name

E) let {legs = 8; eyes = 8; name = "Joe"; spine = false; fur = false}=
 spider in spider.name

F) let {spine=_; name = x} = spider in x
```

- ○ A, D, F
- ○ C, D
- ○ E
- ○ A, B, C, D, F
- ○ A, D
- ○ A, C, D
- ○ None of the above
- ○ A, B, C, D, E

11. Given the following code:

```
let thing = true
let other = ref thing
other := false
```

After these three lines are evaluated, what would the value of `thing`?

- ○ false
- ○ There is a syntax error.
- ○ true

Answers

Q1.1: B
Q1.2: A
Q1.3: D
Q2.1: D
Q2.2: The type of the fold functions expect a list to iterate through, not a variant
Q2.3: C
Q2.4: A,B,C
Q3.1: `Slayp(2, Slayp(3, Slayp(4, Slayp(5, Nfdl))))`
Q3.2: 1,3
Q4.1: C
Q4.2: D
Q5.1: 4
Q5.2: 3
Q6: False
Q7: False
Q8: 18
Q9: `type two = Fish of int, int`
Q10: A, D, F
Q11: true

Lecture Quiz 2/22

1. Here we have a list of strings:

   ```
   aaabbb
   aaaaab
   bbbbbbc
   aaaabc
   ```

   Which of the following regular expressions matches all of the above strings?

   ○ `^a*b*$`
   ○ `^a*b+c?$`
   ○ `^[^abc]*$`
   ○ `^a{3}b{3}$`

2. Which regex is equivalent to `(ab?){1,2}`?

   ○ More than one answer choice
   ○ None of the other answer choices
   ○ `abab|aba|aab|aa|a`
   ○ `abab|aa|a`
   ○ `ab?|(ab?ab?)`

3. Use the following regex for the following questions:
   `^#([0-9][A-Za-z])+ \+ b(0|1)+ = -?[0-9][0-9]*|0x[0-7]{3}$`

   3.1. Would the regex match with the string "`#2F3g + b0 = -4G`" and why or why not?

   ○ No because the string must end with a digit
   ○ Yes
   ○ No because the string it is missing the `0x[0-7]{3}`
   ○ No for some other reason or for multiple reasons listed above
   ○ No because in the part described by `[0-9][A-Za-z]`, the alphabetical characters must all be the same case

   3.2. Would the regex match with the string "`Here is my octal string: 0x777`" and why or why not?

   ○ No for some other reason or for multiple reasons listed above
   ○ No because the ending `777` is not a valid ending
   ○ No because the string must consist only of 3 digits 0-7.
   ○ Yes
   ○ No because there non supported characters in the string

4. Which of the following strings will the following regex NOT accept?
   `^(a(b|c))*d+e?$`

   ○ `ababababdddde`
   ○ `abacabacd`
   ○ `abacabacabacdee`
   ○ `de`

5. Say we're creating a contact form that asks for people's emails and phone numbers. We need help writing regular expressions that can match with them.

5.1. An email begins with a username which can consist of any number of lowercase letters and numbers, as well as any number of "."s in between. The dots just can't be back to back and the username can't begin with a dot. A username must have at least one character.

The following are acceptable examples: `"cmsc330"`, `"cmsc.330"`, `"cmsc.330.cliff"`

We can't accept these: `"cmsc..330"`, `".cmsc330"`, `"CMSC330"`

Next, say the emails we want have an `@` symbol followed by a domain name which must be one of the following three: `"gmail.com"`, `"yahoo.com"`, `"terpmail.umd.edu"`

Which of the following is the right regex for matching emails in this format?

- ○ `^([a-z0-9]+\.?)+@(gmail\.com|yahoo\.com|terpmail\.umd\.edu)$`
- ○ `^([a-zA-Z0-9]+\.?)+@(gmail\.com|yahoo\.com|terpmail\.umd\.edu)$`
- ○ `^([a-z0-9]+\.)+@(gmail\.com|yahoo\.com|terpmail\.umd\.edu)$`
- ○ `^([a-z0-9]+.?)+@(gmail.com|yahoo.com|terpmail.umd.edu)$`
- ○ `^([a-z0-9]+\.?)*@(gmail\.com|yahoo\.com|terpmail\.umd\.edu)$`

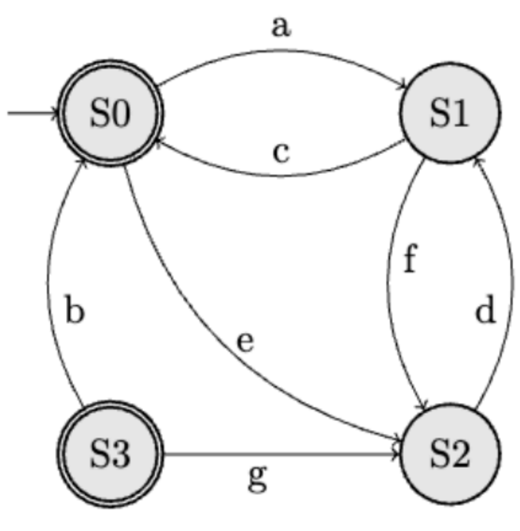
5.2. Let's do phone numbers next. Say we're matching phone numbers in the following format: `(XXX)XXX-XXXX`, where X is any non-negative integer.

Recall that we can use capture groups in regex to extract a select portion of a matched string. We create capture groups by placing parentheses around the portion of the match we'd like to capture.

Which of the following regular expressions will successfully get the area code (only the first 3 digits) as the first capture group?

- ○ `^(\([0-9]{3}\))[0-9]{3}-[0-9]{4}$`
- ○ `^\([0-9]{3}\)[0-9]{3}(-[0-9]{4})$`
- ○ `^\([0-9]{3}\)[0-9]{3}-[0-9]{4}$`
- ○ `^\([0-9]{3}\)([0-9]{3})-[0-9]{4}$`
- ○ `^\(([0-9]{3})\)[0-9]{3}-[0-9]{4}$`

6. Palindromes of all lengths can be represented by a single regular expression.
   - ○ False
   - ○ True
7. An FSM can have multiple start states.
   - ○ False
   - ○ True
8. Use the below image for the next few questions:

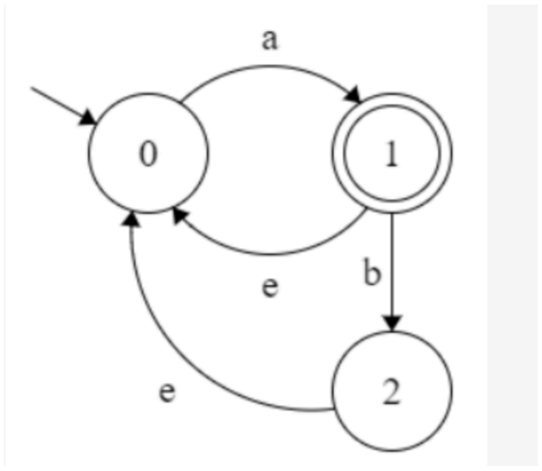


8.1. What state would you end up in if the input string is "`aced`"
- ○ S0
- ○ S2
- ○ S3
- ○ S1
- ○ Garbage/Trash State

8.2. What state would you end up in if the input string is "`baggage`"
- ○ S2
- ○ Garbage/Trash State
- ○ S3
- ○ S1
- ○ S0

8.3. What state would you end up in if the input string is "`afdcedcb`"
- ○ S3
- ○ Garbage/Trash State
- ○ S0
- ○ S1
- ○ S2

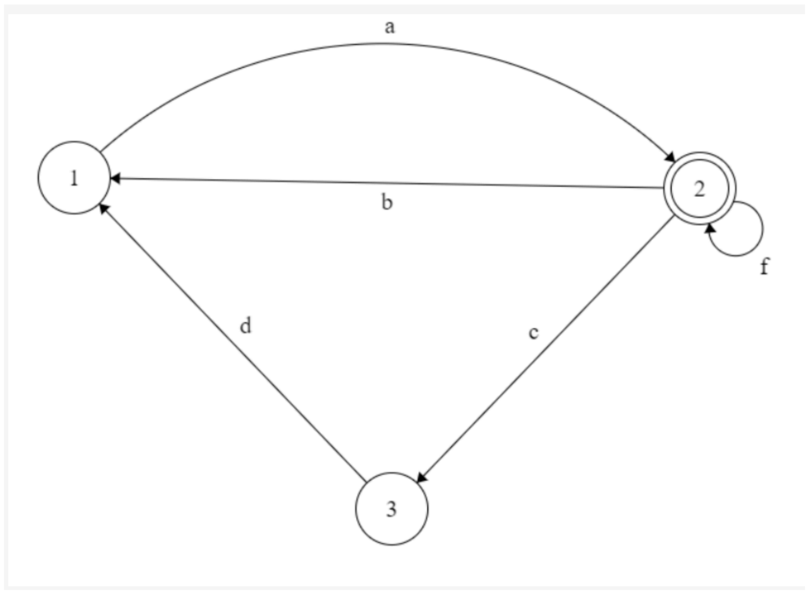
9. Say we're given the following FSM to represent a regex.



What regex does the FSM represent?

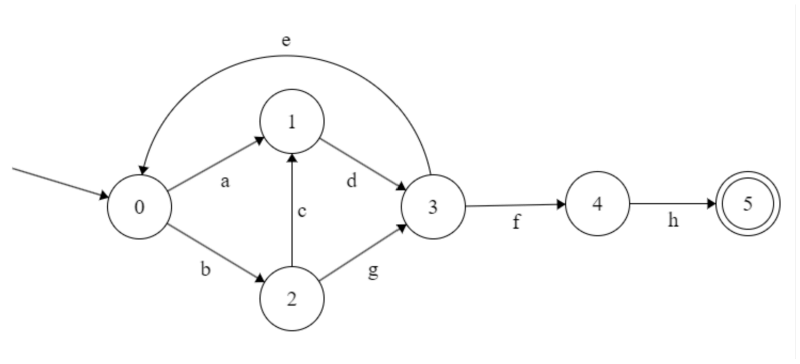
- ○ `a((bea)|(ae))*`
- ○ `a(bea)*`
- ○ `a((bea)|(ea))*`
- ○ `b((bea)|(ea))*`

10. What regex does the following FSM accept (given that the starting state is 1)?



- ○ `af*((b|cd)af*)*`
- ○ `abcdf`
- ○ `af((b|cd)c*)`
- ○ `ab*(f|d)*`

11. Given the above FSM with starting state 0, which of the following strings are accepted?



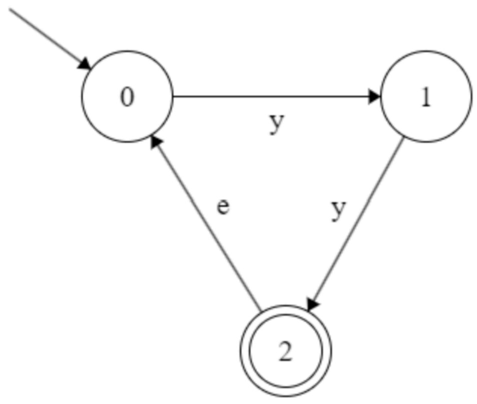
```
A. "adeadfebgfh"
B. "adebcdfh"
C. "bgebgebgef"
D. "adebcdebgfh"
```

- ◯ B, C, D
- ◯ C, D
- ◯ D only
- ◯ A only
- ◯ B, C
- ◯ A, B
- ◯ B, D
- ◯ C only
- ◯ B only

Answers

Q1: `^a*b+c?$`
Q2: `ab?|(ab?ab?)`
Q3.1: Yes
Q3.2: Yes
Q4: `abacabacabacdee`
Q5.1: `^([a-z0-9]+\.?)+@(gmail\.com|yahoo\.com|terpmail\.umd\.edu)$`
Q5.2: `^\(([0-9]{3})\)[0-9]{3}-[0-9]{4}$`
Q6: False
Q7: False
Q8.1: S1
Q8.2: Garbage/Trash State
Q8.3: Garbage/Trash State
Q9: `a((bea)|(ea))*`
Q10: `af*((b|cd)af*)*`
Q11: B, D

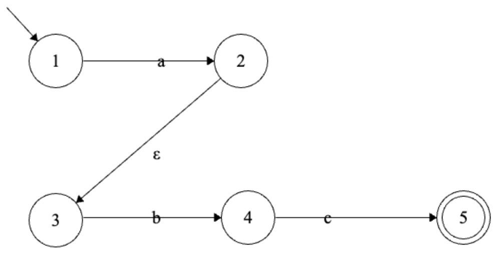
Lecture Quiz 3/1

1. 

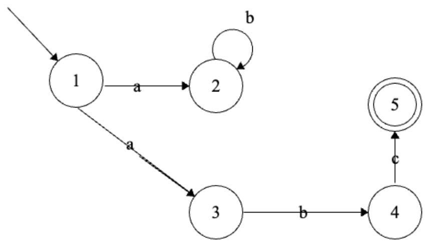
   What is the equivalent regex to the above FSM? (Assume “e” represents an epsilon transition)
   - ○ `(yy)+`
   - ○ `yyyyy*`
   - ○ `y*`
   - ○ `y+`

2. There exists a regular expression that describes mathematical equations in the form $x + y = z$, where $x, y, z \in \mathbb{Z}$
   - ○ True
   - ○ False

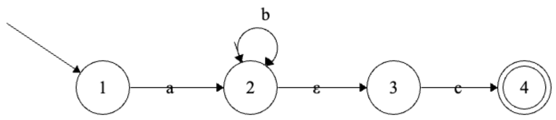
3. Which of the following is NOT an NFA that represents the regex `abc`? (remember that `e` represents an epsilon transition)
   - ○



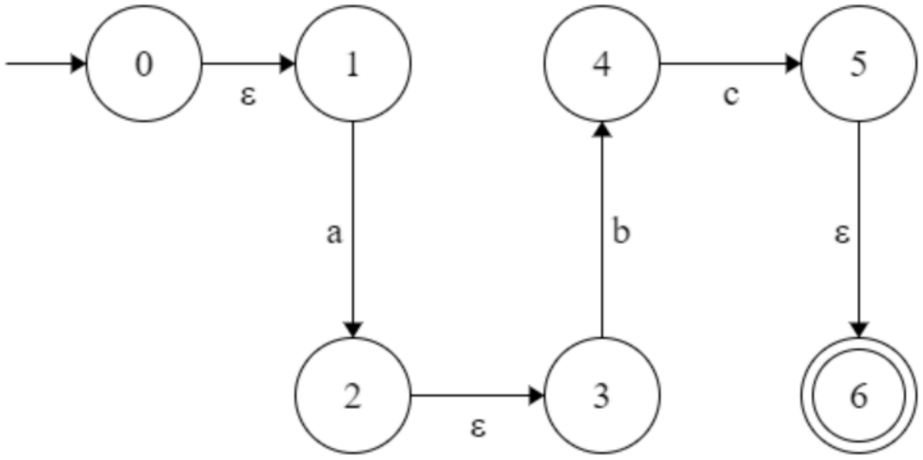
○



○



○



○ More than one of the above do not accept the same strings as the regex `abc`.

○ All of the above accept the same strings as the regex `abc`.

4. The following functions are as defined in Project 3:

   `move nfa lst sym` will output a set (order doesn't matter) of the states that the NFA `nfa` might be in after starting from any state listed in `lst` and making exactly one transition on the symbol `sym` and no other moves.

   `e_closure nfa lst` will output a set (order doesn't matter) of states that the NFA might be in after making zero or more epsilon transitions from any state listed in `lst`.

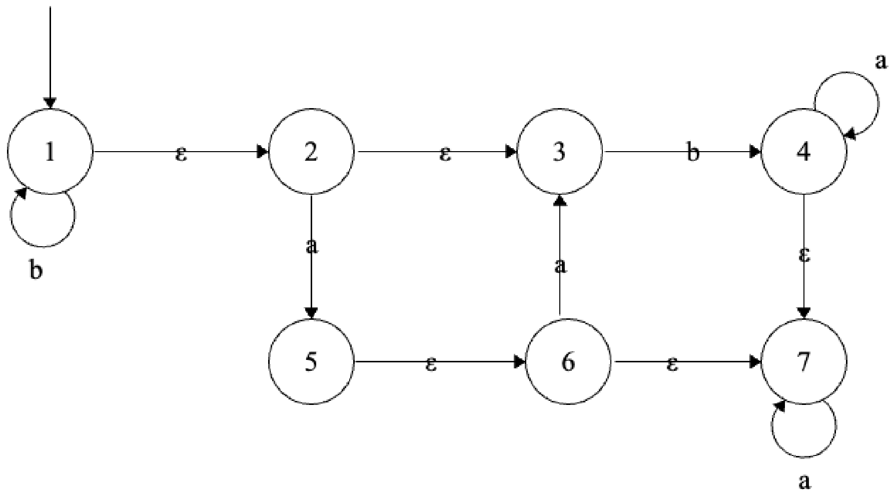
   4.1. The order of `e_closure` and `move` does not matter.

   (ie. `e_closure(move nfa [s] c) = move nfa (e_closure nfa [s]) c` for all arbitrary symbols `c` in the alphabet of the nfa)

   ○ True

   ○ False

   4.2. What would be a valid result of the call `move nfa [2;4;5] "a"` with the following `nfa`? (given that the list `[2;4;5]` represents the set of states 2, 4, and 5, and the string `"a"` represents the symbol a)



   ○ `[4; 5; 6; 7]`

   ○ `[3; 4; 5; 6; 7]`

   ○ None of the listed answers contain the correct states to be returned.

   ○ `[4; 5]`

   ○ `[2; 4; 5]`

4.3. Which of the following would be a valid result of the call `e_closure nfa [2;5]`? `[2; 3; 5; 6; 7]`

- ○ `[2; 3; 5; 6]`
- ○ `[3; 6]`
- ○ None of the listed answers contain the correct states to be returned.
- ○ `[3; 6; 7]`

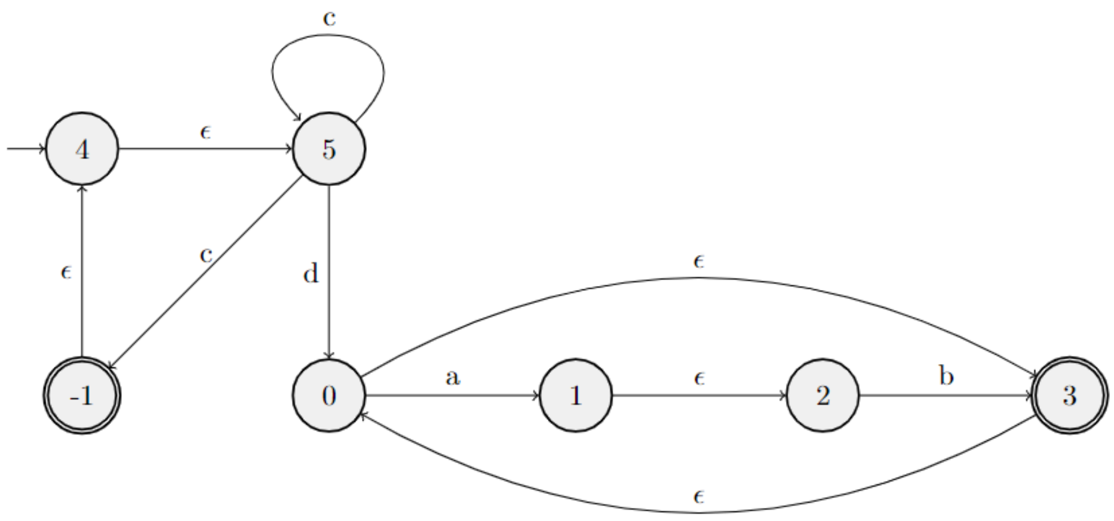
5. Regular Expressions have computational power that is equivalent to Turing Machines.

- ○ False
- ○ True

6. Move and E-closure on a given state will always return the same result

- ○ True
- ○ False

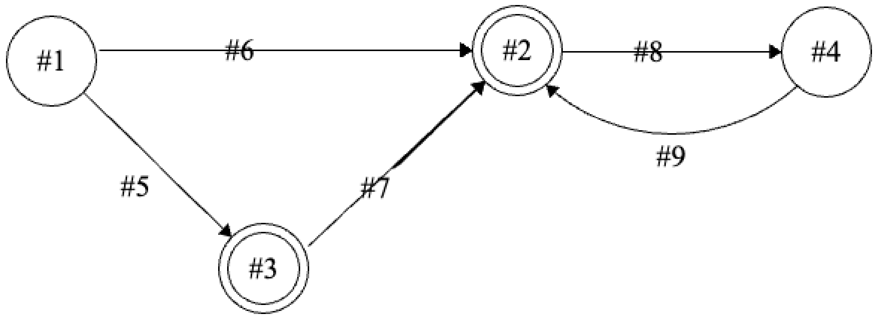
7. Which of the following strings aren't accepted by this NFA?



1. cdc
2. db
3. cda
4. cdb
5. d

- ○ None of the choices
- ○ 3, 4, 5
- ○ 1, 2, 3, 5
- ○ All of the choices
- ○ 1, 2, 3, 4
- ○ 2, 3, 5
- ○ 2, 4

8. Fill in the blanks for the converted DFA based on the above NFA.



8.1. What is blank #1?
- ○ 4,5,-1
- ○ 4
- ○ 4,5
- ○ 0
- ○ 5

8.2. What is blank #2?
- ○ 3,5
- ○ 3,4
- ○ 0,3,4
- ○ 0,3
- ○ 0,4

8.3. What is blank #3?
- ○ 1,5
- ○ 2,3,4,5
- ○ 1,4
- ○ −1,4,5
- ○ 4,5

8.4. What is blank #4?

- ○ 1,6
- ○ 0,3
- ○ 3,5
- ○ 4,6
- ○ 1,2

8.5. What is blank #5?

- ○ a
- ○ ε
- ○ d
- ○ c
- ○ b

8.6. What is blank #6?

- ○ a
- ○ d
- ○ c
- ○ b
- ○ ε

8.7. What is blank #7?

- ○ d
- ○ c
- ○ ε
- ○ b
- ○ a

8.8. What is blank #8?

- ○ c
- ○ a
- ○ b
- ○ ε
- ○ d

8.9. What is blank #9?

- ○ a
- ○ c
- ○ b
- ○ ε
- ○ d

9. Given a NFA with n states, an equivalent DFA can have up to how many states?

- ○ n
- ○ None of the listed options
- ○ lg(n)
- ○ 2^(n)

- ○ n/2

10. When we convert an NFA to a DFA, what is the maximum limit for the number of edges going out a state in the resulting DFA?
    - ○ The limit is the number of original final states in the NFA
    - ○ There is no maximum limit
    - ○ The limit is the number of original states in the NFA
    - ○ None of the listed options
    - ○ The limit is the number of transitions going out of the state in the original NFA
    - ○ The limit is the number of letters in the alphabet in the NFA

Answers

Q1: `(yy)+`
Q2: True
Q3: the one that is straight and has a loop with a b
Q4.1: False
Q4.2: `[4; 5]`
Q4.3: `[2; 3; 5; 6; 7]`
Q5: False
Q6: False
Q7: 1, 2, 3, 4
Q8.1: 4,5
Q8.2: 0,3
Q8.3: −1,4,5
Q8.4: 1,2
Q8.5: c
Q8.6: d
Q8.7: d
Q8.8: a
Q8.9: b
Q9: 2^(n)
Q10: The limit is the number of letters in the alphabet in the NFA

Lecture Quiz 3/8

1. Given the following regex `/a*b+/` which of the following CFG's match the regex?

   (a) S → aS | bT T → ϵϵ | bT

   (b) S → aS | bT | ϵϵ T → b | ϵϵ

   (c) S → aSb | bT T → ϵϵ | bT

   (d) S → Sa | Tb T → ϵϵ | bT

   ○ a
   ○ b
   ○ c
   ○ d

2. Use the following CFG to answer the next few questions:

   S → aS | bT
   T → bT | b | ϵ

   2.1. Which of these strings is allowed by the CFG?

      1. aaaab
      2. abbbb
      3. a
      4. ab
      5. b
      6. ϵ

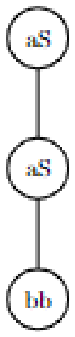
      ○ All of these strings are accepted
      ○ None of these strings are accepted
      ○ 1, 2, 3, 6
      ○ 1, 2, 4, 5
      ○ 2, 3, 4, 6
      ○ 3, 4, 5
      ○ 6
      ○ 4, 5, 6

   2.2. Which regex matches the CFG?

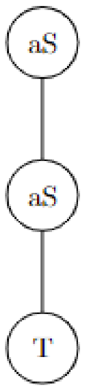
      ○ `a+b?`
      ○ `a*b*`
      ○ `a*b+`
      ○ `a?b+`

   2.3.

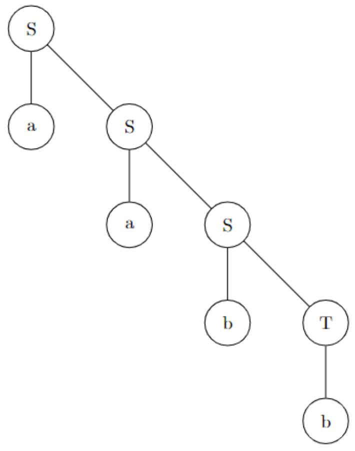
   A.



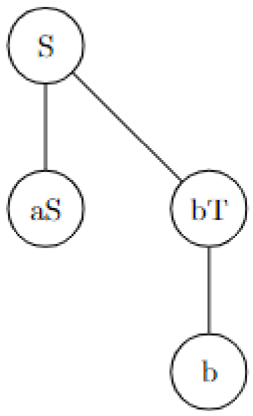
B.



C.



D.



3. For every CFG, there is a matching regular expression.
   - ○ True
   - ○ False

4. The below CFG is ambiguous. We can prove ambiguity by finding two leftmost or two rightmost derivations for the same string.

```
S → S ∧ S|S ∨ S|~S|T
T → true|false
```

   Let's prove this by deriving the string: `true ∧ false ∧ false`

   Here is one valid leftmost derivation:

```
S → S ∧ S
→ S ∧ S ∧ S
→ T ∧ S ∧ S
→ true ∧ S ∧ S
→ true ∧ T ∧ S
→ true ∧ false ∧ S
→ true ∧ false ∧ T
→ true ∧ false ∧ false
```

Select another valid leftmost derivation for the string to prove it's ambiguous:

- ○ S → S ∧ S → S ∧ S ∧ S → S ∧ false ∧ S → true ∧ false ∧ S → true ∧ false ∧ false
- ○ S → S ∧ S → S ∧ S ∧ T → S ∧ S ∧ false → S ∧ false ∧ false → true ∧ false ∧ false
- ○ S → S ∧ S → true ∧ S → true ∧ S ∧ S → true ∧ false ∧ S → true ∧ false ∧ false
- ○ S → S ∧ S → T ∧ S → true ∧ S → true ∧ S ∧ S → true ∧ T ∧ S → true ∧ false ∧ S → true ∧ false ∧ T → true ∧ false ∧ false
- ○ S → S ∧ S → S ∧ S ∧ S → S ∧ S ∧ T → S ∧ S ∧ false → S ∧ T ∧ false → S ∧ false ∧ false → T ∧ false ∧ false → true ∧ false ∧ false

5. Which of the following is not found in a CFG?
   - ○ Terminals
   - ○ Non-Terminals
   - ○ Regex Sequences
   - ○ Production Rules

6. What type of strings does the following CFG accept?

   S → aSb | ab | ϵ

   - ○ The empty string and any word with a or b
   - ○ Words with 1 or more a's followed by 1 or more b's
   - ○ Words with any number of a's followed by any number of b's
   - ○ Words with any number of a's followed by the same number of b's

7. Consider the following CFG:

   S→(S) | ϵ | S) | (S | SS

   7.1. Which of the following is a valid leftmost derivation of the string "(())" for the given grammar?
   - ○ S→(S→((S→((S)→((S))→(())
   - ○ S→SS→S(S)→S(S))→S())→(S())→((S())→(())
   - ○ S→SS→S(S)→(S(S)→(S(S))→(S())→(())
   - ○ None of the above

   7.2. (T/F) The given grammar is not ambiguous.
   - ○ True
   - ○ False

8. Which of the following best describe the same language as the grammar below?

   S → aSUU|T
   T → bTU|ϵ
   U → c|ϵ

Note that the notation $a^x b^y c^z$ denotes a string that has x a's, followed by y b's and z c's. So if we have a notation *axbycz*axbycz where $x = 1, y = 3, z = 2$ this would mean a string like `abbbcc`.

- ○ $a^x b^y c^z$ where $z = x + y$
- ○ $a^x b^y c^z$ where $z \geq x + 2y$
- ○ $a^x b^y c^z$ where $z \leq 2x + y$
- ○ None of the above

9. CFGs can be used to represent palindromes of any length.
   - ○ True
   - ○ False

10. Which of the strings given below are accepted by the following CFG?

    S → S + S|S - S|T
    T → T * T|T / T|N
    N → -N|0|1|2|3|4|5|6|7|8|9

    1. `---9`
    2. `T * T + S`
    3. `2 + 3 * 4 / 5`
    4. `-8`
    5. `-7 / 6 / -5`

    - ○ 1
    - ○ 2
    - ○ 3
    - ○ 4
    - ○ 5
    - ○ 1, 2, 3, 4
    - ○ 1, 3, 4, 5
    - ○ 3, 4, 5
    - ○ 1, 3, 4
    - ○ 1, 2, 3, 4, 5

11. What is the difference between terminals and non-terminals in CFGs?
    - ○ Terminals can be derived further, whereas Non-Terminals cannot.
    - ○ No difference, both Non-Terminals and Terminals can appear in the fully derived string.
    - ○ Non-Terminals represent the actual characters in a string, whereas Terminals represent place-holders.
    - ○ Non-Terminals can be derived further, whereas Terminals cannot.

Answers

Q1: a

Q2.1: 1, 2, 4, 5

Q2.2: `a*b+`

Q2.3: C

Q3: False

Q4: S → S ∧ S → T ∧ S → true ∧ S → true ∧ S ∧ S → true ∧ T ∧ S → true ∧ false ∧ S → true ∧ false ∧ T → true ∧ false ∧ false

Q5: Regex Sequences

Q6: Words with any number of a's followed by the same number of b's

Q7.1: S→(S→((S→((S)→((S))→(())

Q7.2: False

Q8: $a^x b^y c^z$ where $z \leq 2x + y$

Q9: True

Q10: 1, 3, 4, 5

Q11: Non-Terminals can be derived further, whereas Terminals cannot.

Lecture Quiz 3/15

1. Lexer cares about the grammatical meaning of the sentence
   - ○ True
   - ○ False

2. A parser takes in a string as its input and breaks it down the string into a token list
   - ○ True
   - ○ False

3. Every grammar that we make can use the same parser
   - ○ True
   - ○ False

4. What part of the interpreter uses CFGs?
   - ○ Parser
   - ○ Lexer
   - ○ Evaluator

5. Given the following:

   ```
   type token = Tok_Num of char | Tok_Sum
   ```

   What is the result of lexing the following string:

   ```
   7+9++
   ```

   - ○ `[Tok_Num '7'; Tok_Num '9'; Tok_Sum; Tok_Sum; Tok_Sum]`
   - ○ `[Tok_Num '7'; Tok_Sum ;Tok_Num '9'; Tok_Sum; Tok_Sum]`
   - ○ `[Tok_Sum ; Tok_Sum; Tok_Sum; Tok_Num '7'; Tok_Num '9']`
   - ○ None of the above

   For reference, the lexing code is below:

   ```
   type token = Tok_Num of char | Tok_Sum;;

   let lex_string str =
     let rec lex_help pos =
       if pos >= String.length str then
         []
       else
         match str.[pos] with
         | '+' -> Tok_Sum :: lex_help (pos+1)
         | '0' |'1' |'2' |'3' |'4'
         | '5' |'6' |'7' |'8' |'9' ->
            (Tok_Num str.[pos]) :: lex_help (pos+1)
     in
     lex_help 0
   ;;
   ```

6. Consider the following grammar, where n can represent any integer:

```
A -> B * A | B / A | B
B -> C + B | C - B | C
C -> n
```

Also, we have defined a token and ast type for that grammar:

```
type token = Plus | Minus | Star | Slash | Num of int
type ast = Add of ast*ast
         | Sub of ast*ast
         | Mult of ast*ast
         | Div of ast*ast
         | Int of int
```

6.1. For `toks = [Num(1); Plus; Num(2); Star; Num(3); Minus; Num(4)]`, and a parser like the one from lecture, `parse`, what is the output of calling `parse toks`?

- ○ `Mult(Add(Int(1), Int(2)), Sub(Int(3), Int(4)))`
- ○ `Add(Int(1), Mult(Int(2), Sub(Int(3), Int(4))))`
- ○ `Add(Int(1), Sub(Mult(Int(2), Int(3)), Int(4)))`
- ○ There will be an error, as `toks` does not fit the grammar
- ○ None of the above is accurate

6.2. Considering the same grammar as in question 6, which of the following is true if we tried to process the string `1 + 2 * 3 /`

- ○ It would be rejected by the lexer because our grammar can never accept strings that end in /
- ○ It would be rejected by the lexer because the string includes characters that are not valid tokens
- ○ It would be accepted by the lexer, but the resulting token list would be rejected by the parser because our grammar can never accept strings that end in /
- ○ It would be accepted by the lexer, but the resulting token list would be rejected by the parser because the string includes characters that are not valid tokens
- ○ It would cause no errors in either the lexer or the parser
- ○ None of the above

7. Consider the following:

```
type token =  | Tok_Int of int
              | Tok_Plus
              | Tok_Comma

(* instead of matching to figure out the next token,
we can use this instead *)
let lookahead () = match !tok_list with
        | [] -> raise (ParseError "no tokens")
        | (h::t) -> h

(* instead of using a match expression to find the rest of
the token list, you can use this instead *)
let match_tok a = match !tok_list with
        | (h::t) when a = h -> tok_list := t
        | _ -> raise (ParseError "bad match")
```

Which CFG is parsed by the code below?

```
let rec parse_S () =
        parse_T ();
        match lookahead () with
              | Tok_Plus -> (match_tok Tok_Plus; parse_S ())
              | Tok_Comma -> (match_tok Tok_Comma; parse_T ();
 match_tok Tok_Comma; parse_S ())
              | _ -> ()

and parse_T () =
        parse_A ();
        match lookahead () with
              | Tok_Int 7 -> (match_tok (Tok_Int 7))
              | Tok_Int 1415 -> (match_tok (Tok_Int 1415))
              | _ -> ()
and parse_A () =
        match lookahead () with
              | Tok_Int 1715 -> (match_tok (Tok_Int 1715))
              | _ -> ()
```

```
(A)
S -> T + S | T, T, S | T
T -> A 1715 | A 7 | A
A -> 1415 | ε

(B)
S -> S + T | T, T, S | T
T -> A 1715 | A 7 | A
A -> 1415 | ε

(C)
S -> T + T | T, T, S | T
T -> A 1715 | A 7 | A
A -> 1415 | ε

(D)
S -> T + S | T, T, S | T
T -> A 7 | A 1415 | A
A -> 1715 | ε
```

○ A
○ B
○ C
○ D

8. In lecture, it was mentioned that many types of parsers exist. In this class we are using LL(k) parsers. We said the idea was to read left to right when parsing. Could a LL(k) parser (like the one in lecture) parse this?

```
S → SAB| SBA | B | A
A → Aa | a
B → Bb | b
```

○ Yes, because this is a right recursive grammar

◯ Yes, because this is a left recursive grammar
◯ No, because if we read and parse left to right, we get an example of an infinite loop
◯ None of the above

8.1. Determine which grammar describes the same set of strings as the one above and if it can (also?) be parsed with an LL(k) parser.

```
(A)
S → aSAB| bSBA | B | A | ε
A → Aa | a
B → Bb | b

(B)
S → ABS| BAS | BA | AB
A → aA | a
B → bB | b

(C)
S → ASB| BSA | B | A
A → aA | a
B → bB | b
```

◯ A and it can be parsed with an LL(k) Parser
◯ A and it can not be parsed with an LL(k) Parser
◯ B and it can be parsed with an LL(k) Parser
◯ B and it can not be parsed with an LL(k) Parser
◯ C and it can be parsed with an LL(k) Parser
◯ C and it can not be parsed with an LL(k) Parser
◯ Multiple are equivalent but only 1 of those can be parsed with a LL(k) parser
◯ Multiple are equivalent and they can all be parsed with a LL(k) parser
◯ None are equivalent

9. The token types are as follows:

```
type token = Tok_Char of char |  Tok_H | Tok_Mult | Tok_C
(*
  Tok_H for #
  Tok_Mult for *
  Tok_Caret for ^
*)
```

The following functions can be used as a reference to write your parser:

Lookahead Function

```
let lookahead toks = match toks with
   h::t -> h
  | _ -> raise (Failure("Empty input to lookahead"))
```

match_token Function

```
let match_token (toks : token list) (tok : token) : token list =
  match toks with
  | [] -> raise (Failure(string_of_token tok))
  | h::t when h = tok -> t
  | h::_ -> raise (Failure(
      Printf.sprintf "Expected %s from input %s, got %s"
        (string_of_token tok)
        (string_of_list string_of_token toks)
        (string_of_token h)
  ))
```

Consider the following CFG and AST type:

```
S -> cT# | V#
V -> ^T | T
T -> *T|*c
(* c is a character in the alphabet *)
```

```
type ast = Hash of ast | Star of ast | Caret of ast | Char of char | Concat of ast
* ast
```

9.1. Which of the following code snippets is correct when parsing non-terminal **S** and compiles? Note: This returns the rest of the tokens, and then the tree. That is, the type of these functions is `token list -> (token list * ast)`

```
(A)
...
and parse_S toks = match lookahead toks with
  |Tok_Char(x) -> let token_lst_1 = match_token toks Tok_Char in
                  let (token_lst_2,expression1) = parse_T token_lst_1 in
                  (match lookahead token_lst_2 with
                        |Tok_H -> let final_token_lst = match_token token_lst_2
Tok_H in
                                (final_token_lst,Concat(Char(x),Hash(expression_1)))
                      |_ -> failwith "Gotcha lil bro")

  | _ ->         let (token_lst_1,expression_1) = parse_V toks in
                 (match lookahead token_lst_1 with
                        |Tok_H -> let final_token_lst = match_token token_lst_1
Tok_H in
                               (final_token_lst,Concat(Char(x),Hash(expression_1)))
                     |_ -> failwith "Gotcha lil bro")
```

```
(B)
...
and parse_S toks = match lookahead toks with
  |Tok_Char(x) -> let token_lst_1 = match_token toks Tok_Char in
                  let (token_lst_2,expression1) = parse_T token_lst_1 in
                  let final_token_lst = match_token token_lst_2 Tok_H in
                  (final_token_lst,Hash(expression_1))

  |_ ->           let (token_lst_1,expression_1) = parse_V toks in
                  let final_token_lst = match_token Tok_H in
                  (final_token_lst,Hash(expression_1))
```

◯ A
◯ B

◯ Both A and B
◯ Neither A nor

9.2. Which of the following code snippets is correct when parsing non-terminal **V** and compiles?
Which of the following code snippets is correct when parsing non-terminal **V** and compiles?

```
(A)
...
and parse_V toks = let new_token_lst = match_token toks Tok_C in
                   let (new_tok,expression1) = parse_T new_token_lst in
                   (new_tok,Oop(expression1))

(B)
...
and parse_V toks = match lookahead toks with
        |Tok_C -> let new_token_lst = match_token toks Tok_C in
                  let (new_tok,expression1) = parse_T new_token_lst in
                  (new_tok,Caret(expression1))

        | _ ->    let (new_tok,expression1) = parse_T toks in
                  (new_tok,expression1)
```

◯ A
◯ B
◯ Both A and B
◯ Neither A nor B

9.3. Which of the following code snippets is correct when parsing non-terminal **T** and compiles?

```
(A)
...
and parse_T toks = match toks with
                    |Tok_Mult::Tok_Mult::t -> let (new_token_lst,expression1) =
parse_T t in
                                              (new_token_lst,Star(expression1))
                    |Tok_Mult::Tok_Char(x)::t -> (t,Star(Char(x)))
                    |_ -> failwith "Gotcha big bro"

(B)
...
and parse_T toks = let new_token_lst = match_token toks Tok_Mult in
                   (match new_token_lst with
                     |Tok_Mult -> let new_token_lst = match_token new_token_lst
Tok_Mult in
                                   (new_tok,Star(expression1))
                       |Tok_C -> let new_token_lst = match_token new_token_lst
Tok_C in
                                (new_token_lst,Star(Char(x)))
                     | _ -> failwith "Gotcha big bro")
```

◯ A
◯ B
◯ Both A and B

○ Neither A nor B

Answers

Q1. False
Q2. False
Q3. False
Q4. Parser
Q5: `[Tok_Num '7'; Tok_Sum ;Tok_Num '9'; Tok_Sum; Tok_Sum]`
Q6.1: `Mult(Add(Int(1), Int(2)), Sub(Int(3), Int(4)))`
Q6.2: It would be accepted by the lexer, but the resulting token list would be rejected by the parser because our grammar can never accept strings that end in `/`
Q7: D
Q8: No, because if we read and parse left to right, we get an example of an infinite loop[1]
Q8.1: None are equivalent
Q9.1: Both A and B
Q9.2: B
Q9.3: Neither A nor B

[1]The answer is probably this, but the true answer was not revealed, and the question was impossible to answer on the quiz itself

Lecture Quiz 3/29

1. Operational Semantics describe meanings through how things execute
   - ○ True
   - ○ False

2. CFGs are to Parsers as OpSem is to ____
   - ○ Lexers
   - ○ Interpreters
   - ○ Tokenizers
   - ○ Parsers

3. What is the purpose of environments in OpSem?
   - ○ To evaluate the target language
   - ○ To parse the target language
   - ○ To provide us with natural resources
   - ○ To store the bindings of variables

4. Take the following rules:

$$\frac{}{A;\ \text{true} \Rightarrow \text{true}} \quad \frac{}{A;\ \text{false} \Rightarrow \text{false}}$$

$$\frac{A;\ e_1 \Rightarrow \text{true}}{A;\ (\text{not } e_1) \Rightarrow \text{false}} \quad \frac{A;\ e_1 \Rightarrow \text{false}}{A;\ (\text{not } e_1) \Rightarrow \text{true}}$$

$$\frac{A;\ e_1 \Rightarrow \text{true} \qquad A;\ e_2 \Rightarrow v_1}{A;\ (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \Rightarrow v_1}$$

$$\frac{A;\ e_1 \Rightarrow \text{false} \qquad A;\ e_3 \Rightarrow v_1}{A;\ (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \Rightarrow v_1}$$

$$\frac{A;\ e_1 \Rightarrow v_1 \qquad A;\ e_2 \Rightarrow v_2 \qquad v_3 \text{ is } v_1 \mid\mid v_2}{A;\ (e_1 \mid\mid e_2) \Rightarrow v_3}$$

4.1. In the above rules, A; true => true and A; false => false, are what type of rule:

○ Hypothesis
○ Expression
○ Mapping
○ Axiom

4.2. In the above rules, what is the difference between terms containing “e”, such as e1 or e2 and temrs containing “v”, such as v1 or v2?

○ e refers to a binding from the environment and v refers to the value of the binding
○ e refers to expressions and v refers to values that an expression evaluates to

○ e refers to a binding from the environment and v refers to another OpSem rule

5. Given the rules from Problem 5, and the OpSem proof below, fill in the blanks accordingly

$$\dfrac{\dfrac{\dfrac{}{A;\ \#3 \Rightarrow \#3} \qquad \dfrac{}{A;\ \#4 \Rightarrow \#4} \qquad \#5 \text{ is} \#1}{A;\ \#1 \Rightarrow \#5} \qquad \dfrac{}{A;\ \#2 \Rightarrow \#2}}{\text{if } true || false \text{ then } true \text{ else not} false \Rightarrow \#6}$$

5.1. Blank 1:
○ true||false
○ true
○ false
○ if true then true else not false

5.2. Blank 2:
○ false
○ true
○ true||false => true
○ (not false) => true

5.3. Blank 3:
○ (not false)
○ false
○ true||false
○ true

5.4. Blank 4:
○ false
○ true
○ (not true)
○ (not false)

5.5. Blank 5:
○ (not true)
○ if false then true else false
○ false
○ true

5.6. Blank 6:
○ true
○ false
○ (not true)
○ (not false)

6. A property is something we expect to be true about our code
○ True
○ False

7. The following is a valid property: The union of two sets should contain all the distinct elements of both sets
   - ○ True
   - ○ False
8. Here is a function we want to test that contains a bug:

   ```
   (* this function is supposed to reverse a list *)
   let reverse lst = fold_left (fun a x -> a @ [x]) [] lst
   ```

   Here is a property we are testing:

   > The length of any list should be equal to the length of the reversed list

   This property will catch a bug in the above implementation

   - ○ False
   - ○ True
9. Suppose we want to test the following property about mergesort.

   > After sorting a list, the minimum value should be the first one.

   Does the following function describe this property correctly?

   ```
   fun x -> min (mergesort x) = List.hd (x)
   ```
   - ○ False
   - ○ True
10. Suppose we are testing an implementation of the preorder traversal of a tree.

    > in a preorder traversal of a tree, the first element of the list should be the root of the tree

    Does the following function describe this property correctly?

    ```
    fun x -> match x with
       | Node(value, left, right) -> value = List.hd (preorder x)
    ```

    - ○ True
    - ○ False
11. Suppose we want to test the following property about FSMs:

    > If you convert an NFA to a DFA, the same strings should be accepted by both machines

    Would this code test this property correctly?

    ```
    fun fsm str -> if accept(str,fsm) then accept(str, nfa_to_dfa fsm) else true
    ```

    - ○ False
    - ○ True
12. For a binary search tree, inorder traversal of the nodes should be the sorted list of the values. Which function tests this property given that `tree` is the BST and `lst` is a list of the values, and `mergesort` correctly returns the sorted list?
    - ○ `fun tree lst -> (inorder tree) = lst`

○ `fun tree lst -> (inorder tree) = (mergesort lst)`
○ `fun tree lst -> tree = (mergesort lst)`
○ `fun tree lst -> tree = lst`

13. Suppose we want to test the following property about binary trees.

    > after inserting a node into a tree, that value should exist in the tree.

    The following is a valid function that describes this property:

    ```
    fun val tree -> exists(val,tree)
    ```

    ○ False
    ○ True

Answers

Q1: False
Q2: True
Q3.1: “ha”
Q3.2: true
Q3.3: false
Q3.4: G, x : string
Q3.5: string
Q3.6: x^x : string
Q3.7: ^ = (string, string, string)
Q4.1: No
Q4.2: Yes
Q4.3: No
Q5: Yes
Q6.1: No
Q6.2: No
Q6.3: Yes
Q6.4: Yes
Q6.5: No

Lecture Quiz 4/5

1. Static type systems are also always complete.
   - ○ True
   - ○ False
2. Well-typed languages are not always well-defined.
   - ○ True
   - ○ False
3. Use the following type checking rules for the next few questions.

$$\frac{}{G \vdash \text{true} : \text{bool}} \qquad \frac{}{G \vdash \text{false} : \text{bool}} \qquad \frac{}{G \vdash \text{“ha”} : \text{string}}$$

$$\frac{}{G \vdash \text{x} : G(\text{x})} \qquad \frac{G \vdash e_1 : \text{string} \quad G \vdash e_2 : \text{string} \quad \hat{} = (\text{string}, \text{string}, \text{string})}{G \vdash e_1 \,\hat{}\, e_2 : \text{string}}$$

$$\frac{G \vdash e_1 : \text{bool} \quad G \vdash e_2 : \text{bool} \quad || = (\text{bool}, \text{bool}, \text{bool})}{G \vdash e_1 \,||\, e_2 : \text{bool}}$$

$$\frac{G \vdash e_1 : t_1 \quad G, x : t_1 \vdash e_2 : t_2}{G \vdash \mathit{let}\ x = e_1 \text{ in } e_2 : t_2} \qquad \frac{G \vdash e_1 : \text{bool} \quad G \vdash e_1 : t \quad G \vdash e_1 : t}{G \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : t}$$

$$\frac{\dfrac{}{G \vdash \textbf{[1]} : \text{string}} \quad \dfrac{\dfrac{\dfrac{}{\textbf{[4]} \vdash \textbf{[2]} : \text{bool}} \quad \dfrac{}{\textbf{[4]} \vdash \textbf{[3]} : \text{bool}} \quad || = (\text{bool}, \text{bool}, \text{bool})}{\textbf{[4]} \vdash \text{true} \,||\, \text{false} : \text{bool}} \quad \dfrac{}{\textbf{[4]} \vdash x : \text{string}} \quad \dfrac{\dfrac{}{\textbf{[4]} \vdash x : \textbf{[5]}} \quad \dfrac{}{\textbf{[4]} \vdash x : \textbf{[5]}} \quad \textbf{[7]}}{\textbf{[4]} \vdash \textbf{[6]}}}{\textbf{[4]} \vdash \mathit{if}\ \text{true} \,||\, \text{false}\ \mathit{then}\ x\ \mathit{else}\ x\hat{}x : \text{string}}}{G \vdash \mathit{let}\ x = \text{“ha” in } \mathit{if}\ \text{true} \,||\, \text{false}\ \mathit{then}\ x\ \mathit{else}\ x\hat{}\ x : \text{string}}$$

   3.1. What should replace [1]?
   - ○ true
   - ○ false
   - ○ bool
   - ○ string
   - ○ x
   - ○ “ha”

   3.2. What should replace [2]?
   - ○ true
   - ○ false
   - ○ bool
   - ○ string
   - ○ x
   - ○ “ha”

   3.3. What should replace [3]?
   - ○ true
   - ○ false
   - ○ bool

○ string
○ x
○ “ha”

3.4. What should replace [4]?
○ G
○ G, x : string
○ G, x : bool
○ G, x : “ha”

3.5. What should replace [5]?
○ true
○ false
○ bool
○ string
○ x
○ “ha”

3.6. What should replace [6]?
○ “ha” : string
○ true || false : bool
○ x^x : string
○ x^x : “haha”

3.7. What should replace [7]?
○ || = (bool, bool, bool)
○ x : bool
○ x : string
○ ^ = (string, string, string)

4. Using the same rules as above,

$$\frac{}{G \vdash \text{true} : \text{bool}} \qquad \frac{}{G \vdash \text{false} : \text{bool}} \qquad \frac{}{G \vdash \text{“ha”} : \text{string}}$$

$$\frac{}{G \vdash \text{x} : G(\text{x})} \qquad \frac{G \vdash e_1 : \text{string} \quad G \vdash e_2 : \text{string} \quad \text{^} = (\text{string, string, string})}{G \vdash e_1 \text{ ^ } e_2 : \text{string}}$$

$$\frac{G \vdash e_1 : \text{bool} \quad G \vdash e_2 : \text{bool} \quad || = (\text{bool, bool, bool})}{G \vdash e_1 \,||\, e_2 : \text{bool}}$$

$$\frac{G \vdash e_1 : t_1 \quad G, x : t_1 \vdash e_2 : t_2}{G \vdash \textit{let } x = e_1 \text{ in } e_2 : t_2} \qquad \frac{G \vdash e_1 : \text{bool} \quad G \vdash e_1 : t \quad G \vdash e_1 : t}{G \vdash \textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3 : t}$$

answer the following:

4.1. Will the following pass the type check enforced by the rules above? (as usual, the “x” in the rules is a stand-in for any given variable)

```
let var = true in if var then true else "ha"
```

○ Yes

○ No

4.2. Will the following pass the type check enforced by the rules above?

```
let var = true in if var || true then false else false
```

○ Yes

○ No

4.3. Will the following pass the type check enforced by the rules above?

```
"haha"
```

○ Yes

○ No

5. Given the following rules:

$$\frac{G \vdash e_1 : bool \qquad G \vdash e_2 : number \qquad G \vdash e_3 : number}{G \vdash if\ e_1\ then\ e_2\ else\ e_3 : number}$$

$$\frac{G \vdash e : int \qquad int <: number}{G \vdash e : number} \qquad \frac{G \vdash e : float \qquad float <: number}{G \vdash e : number}$$

$$\frac{}{G \vdash true : bool} \qquad \frac{}{G \vdash false : bool}$$

$$\frac{}{G \vdash n : int} \qquad \frac{}{G \vdash f : float}$$

Where n is any integer and f is any float.

Will the following pass the type check enforced by the rules above?

```
if true then 3 else 4.0
```

○ Yes

○ No

6. Considering the OCaml programming language, answer the following subtype questions

6.1. Is the following true:

```
{x:float} <: {y:float}
```

○ Yes

○ No

6.2. Is the following true:

```
{x:int; y:float} <: {x:float}
```

○ Yes

○ No

6.3. Is the following true:

```
{x:float; y:bool} <: {y:bool}
```

○ Yes

○ No

6.4. Is the following true:

```
{x:{a:float;b:int}; y:{c:string}} <: {x:{};y:{c:string}}
```

○ Yes

○ No

6.5. Is the following true:

```
{x:{a:float;b:int}; y:{c:string}} <: {x:{b:int};y:{d:string}}
```

○ Yes

○ No

Answers

Q1: False
Q2: True
Q3.1: “ha”
Q3.2: true
Q3.3: false
Q3.4: G, x : string
Q3.5: string
Q3.6: x^x : string
Q3.7: ^ = (string, string, string)
Q4.1: No
Q4.2: Yes
Q4.3: No
Q5: Yes
Q6.1: No
Q6.2: No
Q6.3: Yes
Q6.4: Yes
Q6.5: No

Lecture Quiz 4/12

1. Here's a lambda calc expression:

   `(λc. (λb. a) (b a)) (λx. (λx. (λy. x y)) x)`

   1.1. In the above lambda calc expression, which variables are free?
   - ○ a, b
   - ○ a, b, c
   - ○ a, b, c, x, y
   - ○ c, b, y
   - ○ c, b, x, y

   1.2. Which of the following is the correct Beta Normal Form of the above lambda calc expression?
   - ○ `(λx. (λx. (λy. y)))`
   - ○ `a`
   - ○ `b (λx. (λx. (λy. y)))`
   - ○ `b a`

2. Which of the following properly uses parentheses to make the implicit associativity and operations explicit for the following lambda calc expression? **using maximum amount of parentheses**

   `λb. λc. a c λa. b a`

   - ○ `(λb. (λc. (a (c (λa. (b a))))))`
   - ○ `(λb. (λc. ((a c) (λa. (b a)))))`
   - ○ `(λb. (λc. (a (c λa. b a))))`
   - ○ `(λb. λc.) (a (c λa. (b a)))`

3. Which of the following OCaml functions is the same as the following lambda calc expression: `λa. λb. a b (λc. c)`
   - ○ `fun a b -> fun c -> c`
   - ○ `fun a b c -> a b c`
   - ○ `fun a -> fun b -> a b (fun c -> c)`
   - ○ `fun a -> fun b -> a b c`

4. Which of the following lambda calc expression accurately represents the following OCaml expression: `a (fun b -> a b) c (fun d b -> d)`
   - ○ `a (λb. a b) c (λd. λb. d)`
   - ○ `a λb. a b c (λd. λb. d)`
   - ○ `a (λb. a b) c (λd b. d)`
   - ○ `a λb. a b c (λd b. d)`

5. Which expression is alpha equivalent to `(λb. (λb. b)) (λc. (λa. c))`?
   - ○ `(λb. (λd. c)) (λc. (λa. a))`
   - ○ `(λe. (λd. d)) (λa. (λc. a))`
   - ○ `(λb. (λd. b)) (λc. (λa. c))`
   - ○ `(λe. (λd. c)) (λa. (λc. c))`

6. Given the following lambda calc expression, `b ((λa. a b) (λc. a c))`, answer the following questions.

6.1. How many beta reductions are needed to reduce `b ((λa. a b) (λc. a c))` to beta normal form?

- ○ 1
- ○ 2
- ○ 3
- ○ 4
- ○ 5
- ○ Not possible to reduce to beta normal form

6.2. What is the beta normal form of `b ((λa. a b) (λc. a c))`?

- ○ `b ((λc. a c) b)`
- ○ `b (a (λa. a b))`
- ○ `b a`
- ○ `b (a b)`
- ○ `b a b`
- ○ Not possible to reduce to beta normal form

7. Given the following lambda calc expression, `a (((λx. x a (λy. x y a)) b) (λf. f))`, answer the following questions.

7.1. How many beta reductions are needed to reduce `a (((λx. x a (λy. x y a)) b) (λf. f))` to beta normal form?

- ○ 1
- ○ 2
- ○ 3
- ○ 4
- ○ 5
- ○ Not possible to reduce to beta normal form

7.2. What is the beta normal form of `a (((λx. x a (λy. x y a)) b) (λf. f))`?

- ○ `a b a (λy. b y a) (λf. f)`
- ○ `a b a b (λf. f) a`
- ○ `a (λx. x a x b a) (λf. f)`
- ○ `a (b a (λy. b y a)(λf. f))`
- ○ `a b a b a`
- ○ `a a b a`

7.3. Which expressions are alpha equivalent to `a (((λx. x a (λx. x y a)) b) (λf. f))`?

```
A. b (((λx. x b (λe. x e b)) b) (λf. f))
B. a (((λx. x a (λf. f y a)) b) (λg. g))
C. a (((λx. x a (λy. x x a)) b) (λf. f))
D. a (((λe. e a (λt. t y a)) b) (λg. g))
```

- ○ A, C
- ○ A
- ○ B
- ○ C

- ○ B, D
- ○ D
- ○ None of the choices
- ○ All of the choices

Answers

Q1.1: a,b

Q1.2: `a`

Q2: `(λb. (λc. ((a c) (λa. (b a)))))`

Q3: `fun a -> fun b -> a b (fun c -> c)`

Q4: `a (λb. a b) c (λd. λb. d)`

Q5: `(λe. (λd. d)) (λa. (λc. a))`

Q6.1: 2

Q6.2: `b (a b)`

Q7.1: 1

Q7.2: `a (b a (λy. b y a)(λf. f))`

Q7.3: B, D

Lecture Quiz 4/19

1. What is the result of performing a single lazy evaluation on the following lambda calc expression?

   `(λc. (λa. (λd. c))) ((λx. y) x)`

   - ○ `(λc. (λa. (λd. c))`
   - ○ `(λc. (λx. y x))`
   - ○ `(λa. (λd. ((λx. y) x))`
   - ○ `(λx. y x)`

2. What is the result of performing a single eager evaluation on the following lambda calc expression? `(λc. (λb. b)) ((λa. c) (b a))`

   - ○ `(λb. b)`
   - ○ `(λc. (λb. b))(c)`
   - ○ `(λc. (λb. b))(b a c)`
   - ○ None of the above

3. What is the beta-normal form for the following lambda expression? `(λc.((λb.b)(λc.b)))(λb.(λa.(bb)))`

   - ○ `λc.b`
   - ○ `λc.bb`
   - ○ `λc.aa`
   - ○ None of the above

4. Given a lambda calc expression, the result of a single eager evaluation and a single lazy evaluation will always be the same.
   - ○ True
   - ○ False

5. Given a lambda calc expression, if I reduce it down to beta-normal Form using eager evaluation, and reduce it down to beta-normal Form using lazy evaluation, the results will be the same.
   - ○ True
   - ○ False

6. The following lambda expression may or may not need to be alpha-converted to evaluate correctly: `((λa. λb. a b) b) c`

   What is the correct beta-normal form of this expression?

   - ○ a b
   - ○ b b
   - ○ b c
   - ○ c c

7. Which of the following lambda expressions is not alpha equivalent to the others?

   - ○ λg.g λy.y c λy.y y
   - ○ λx.x λy.y c λz.z y

- ◯ λv.v λh.h c λo.o h
- ◯ They are all alpha equivalent to each other.

8. Garbage collection is always handled by a language by itself
   - ◯ False
   - ◯ True
9. Stack supports automatic allocation and deallocation of data
   - ◯ True
   - ◯ False
10. It is necessary to deal with the removal of data properly because issues can potentially lead to:
    - ◯ Misuse of data
    - ◯ Unauthorized access to secured files
    - ◯ File content leak
    - ◯ All of the above
11. Reference copying is a type of garbage collector that can be used to handle cyclic data structures
    - ◯ False
    - ◯ True
12. Fragmentation in memory doesn't affect a user's access to contiguous memory space
    - ◯ False
    - ◯ True
13. Mark and Sweep Garbage collection method overcomes the problem of Fragmentation
    - ◯ False
    - ◯ True
14. Answer the following Church Encoding questions.
    - 14.1. Which of these are encodings for `and`?
      - ◯ `λy. λx. y y x`
      - ◯ `λx. λy. x y x`
      - ◯ `λx. λy. x x x`
      - ◯ `λx. λy. x y y`
    - 14.2. Which of these are encodings for `or`?
      - ◯ `λy. λx. y y x`
      - ◯ `λx. λy. x y x`
      - ◯ `λx. λy. x x y`
      - ◯ `λx. λy. x y y`
    - 14.3. What is the encoding for `not`?
      - ◯ `λy. λx. false y true`
      - ◯ `λx. λy. x y true`
      - ◯ `λx. λy. false x y`
      - ◯ `λx. x false true`
    - 14.4. What is the encoding for XOR?

- ◯ `λx. λy. y (not y) x`
- ◯ `λx. λy. x (not y) y`
- ◯ `λx. λy. x (not x) y`
- ◯ `λx. λy. y (not x) y`

15. ***BONUS*** Consider the following encodings:

```
if a then b else c: a b c
0 : λf.λx.x
1 : λf.λx.f x
plus1: λn.λf.λx.f (n f x)
iszero: λn.n (λx.λy.y) (λx.λy.x)
```

What OCaml expression represents the following lambda calc expression:

```
(λn.n (λx.λy.y) (λx.λy.x) (λf.λx.f x)) (λf.λx.f x) ((λn.λf.λx.f (n f x)) (λf.λx.x))
```

- ◯ `if (iszero 1) then 1 else (plus1 0)`
- ◯ `if (iszero 0) then 0 else (plus1 1)`
- ◯ `if (iszero 1) then 0 else (plus1 1)`
- ◯ `if (iszero 0) then 0 else (plus1 0)`
- ◯ `none of the other options`

Answer

Q1: `(λa. (λd. ((λx. y) x))`
Q2: `(λc. (λb. b))(c)`
Q3: `λc.b`
Q4: False
Q5: True
Q6: b c
Q7: λg.g λy.y c λy.y y
Q8: False
Q9: True
Q10: All of the above
Q11: False
Q12: False
Q13: False
Q14.1: `λx. λy. x y x`
Q14.2: `λx. λy. x x y`
Q14.3: `λx. x false true`
Q14.4: `λx. λy. x (not y) y`
Q15.5: `if (iszero 1) then 1 else (plus1 0)`

Lecture Quiz 4/26

1. True or False: By default, things are mutable in Rust
   - ○ True
   - ○ False

2. What is the type of this expression

```
{
let x = 32;
let y = true;
let z = if y{x; 1.0} else {let a = 2.0; a}
}
```

   - ○ f32
   - ○ unit
   - ○ f64
   - ○ None of the above

3. Consider the following Rust code:

```
fn main () {
    let z = {
        let x = 56;
        let y = 38;
        y = x + 12;
        x + y
    };
    print!("{}", z);
 }
```

   What will be the output? (Be careful about the declarations and changes to variables)

   - ○ 124
   - ○ 94
   - ○ will not compile
   - ○ 106

4. Given the following code

```
(1) let cliff = 83;
(2) let lysine = String :: from("no thanks");
(3) let lolcode = lysine;
(4) let lysine = cliff;
```

   4.1. How many instances of `83` are there and who owns them after line 4?
   - ○ 1; `cliff`
   - ○ 1; `lysine`
   - ○ 2; `lysine`, `cliff`
   - ○ None

   4.2. How many instances of `"no thanks"` are there and who owns them after line 3?
   - ○ 1; `lolcode`
   - ○ 1; `lysine`

- ◯ 2; `lolcode`, `lysine`
- ◯ None

4.3. Here is a modified version of the code above:

```
let lysine = String::from("no thanks");
let lolcode = lysine.clone();
```

What best describes the variables in this code block?

- ◯ `lolcode` is now the owner of `"no thanks"`
- ◯ `lolcode` would be borrowing `"no thanks"` from `lysine`. `lysine` is the owner of `"no thanks"`
- ◯ `lolcode` and `lysine` are both owners of separate instances of `"no thanks"`
- ◯ None of the above

5. Consider the code below:

```
(1) let a = String::from("rust is fun");
(2) let b = a;
(3) let c = &b;
```

5.1. After the execution of line 2, what variable owns the string “rust is fun”?

- ◯ a
- ◯ b
- ◯ c

5.2. After the execution of line 3, what variable owns the string “rust is fun”?

- ◯ a
- ◯ b
- ◯ c

6. In Rust, a piece of data can have any number of both mutable and immutable references.

- ◯ True
- ◯ False

7. Consider the following rust code:

```
fn main() {
    let a = String::from("ab");
    let b = String::from("cd");
    //Print 1
    println!("{}{}", a,b);

    let b = a;
    let a = f();
    //Print 2
    println!("{}{}", a,b);

    let a = g(b);
    let b = g(a);
    let a = b;
    let b = f();
    //Print 3
    println!("{}{}", a,b);
}

fn f() -> String {
    String::from("ef")
}

fn g(a:String) -> String {
    a
}
```

7.1. What will be printed after Print 1?
- ◯ abcd
- ◯ abab
- ◯ cdcd
- ◯ cdab

7.2. What will be printed after Print 2?
- ◯ efef
- ◯ cdab
- ◯ abcd
- ◯ efab

7.3. What will be printed after Print 3?
- ◯ feba
- ◯ abef
- ◯ abab
- ◯ fefe

8. Which of the following are properties that define Rust's ownership rule?

```
1. Each value in Rust has an owner
2. there can be multiple owners of a value at the same time
3. When the owner goes out of scope, the value will be dropped
4. While a value has an owner, it may not be accessed by anything
 inside and outside of its scope
```

- ◯ 3, 4
- ◯ 1, 2, 3
- ◯ 1, 3
- ◯ 2, 4
- ◯ 1, 2, 3, 4
- ◯ None of the choices

9. True or False: Rust prevents double freeing of memory.
- ◯ True
- ◯ False

10. What is the type of the following expression:

```
{
let x = 32;
let y = true;
let z = if y {x; 1.0} else {let a = 2.0; a};
z
}
```

- ○ u32
- ○ f64
- ○ i32
- ○ i64

Answers

Q1. False
Q2. None of the above
Q3. will not compile
Q4.1. 2; lysine, cliff
Q4.2. 1; lolcode
Q4.3. lolcode and lysine as both owners of seperate instances of “no thanks”
Q5.1. b
Q5.2. b
Q6. False
Q7.1. abcd
Q7.2. efab
Q7.3. abef
Q8. 1, 3
Q9. True
Q10. f64

Lecture Quiz 5/3

1. Suppose `x` has type `&'a i32`. `'a` refers to x's scope
   - ◯ False
   - ◯ True
2. We can have as many mutable references as we want in Rust.
   - ◯ False
   - ◯ True
3. When a borrowed variable goes out of scope the data associated with the variable gets dropped.
   - ◯ False
   - ◯ True
4. A function will always gain ownership of any and all data passed to it through a parameter.
   - ◯ False
   - ◯ True
5.

```
{
  let mut x = String::from("Hello");                           (1)
    let z = {
        let s2 = &x;                                           (2)
        println!("{} == {}", x, s2);                           (3)
        x.push_str(" World");                                  (4)
    };
  println!("{}", x);                                           (5)
}
```

   Answer the following questions based on the above code

   5.1. The code violates the rule of having either one mutable reference or infinitely many immutable references and thus won't compile.
   - ◯ False
   - ◯ True

   5.2. The `println` in line `(3)` prints out the following:
   - ◯ `Hello == World`
   - ◯ `Hello == Hello`
   - ◯ `Hello === xyz` where xyz is the memory address of `x`
   - ◯ Compile Error
   - ◯ None of the above

   5.3. The `println` in line `(5)` prints out the following:
   - ◯ `World`
   - ◯ `Hello World`
   - ◯ `Hello`
   - ◯ Compile Error
   - ◯ None of the above
6. Consider the following Rust code:

```
fn main() {
    let x = String::from("Hello");                             (1)
    let y = x;                                                 (2)
    let z = hehe(y);                                           (3)
    println!("{}", z);                                  (4)
}

fn hehe(w: String) -> i64 {
    4                                                   (5)
}
```

What happens here?

○ Line 2 creates a mutable reference to `x` while line 4 tries to use it as an immutable reference, causing a compiler error.

○ The function call on line 3 causes the parameter `w` of the function `hehe` to gain ownership of "Hello", causing a compiler error when line 4 tries to print `z`.

○ Line 3 causes `z` to gain ownership of "Hello", and so "Hello" is successfully printed.

○ Line 3 sets the value of `z` to `4`, and so "4" is successfully printed.

7. Will the following compile? If not, what's an issue?

```
(1)    let mut x = String::from("Hello");
(2)    let y = &mut x;
(3)    x.push_str(" World");
(4)    y.push_str(" World");
```

○ It will compile.

○ You are not allowed to create a mutable reference to mutable data, as this violates the "only one mutable reference" rule.

○ Since `y` is a mutable reference, `x` cannot be used as a mutable reference again until the lifetime of `y` is over.

○ `push_str` can only be called using an immutable reference.

8. Will the following compile? If not, what's an issue?

```
fn main() {
  let ve = make_vec();
  println!("ve[0]: {}",ve[0]);
}

fn make_vec() -> Vec<&String>{
  let s = String::from("Hello");
  let mut v = vec![];
  v.push(&s);
  return v;
}
```

○ It will compile.

○ The lifetime of `s` ends at the end of `make_vec`, and so the pushed `&s` would become a dangling pointer once the vector is returned.

○ `ve` is not declared as mutable, while `v` in `make_vec` is, and so you cannot set `ve` to `v`.

○ `&s` is of type &str and not &String, and so cannot be pushed to a vector that expects values of type &String.

9. Will the following compile? If not, what's an issue?

```rust
fn longer(a:&String, b:&String) -> &String {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}

fn main() {
    let x = String::from("Hello");
    let y = x.clone();
    let z = longer(&x,&y);
    println!("{}", z);
}
```

- ◯ It will compile.
- ◯ y gains ownership of "Hello", and so x can no longer be used after y is set.
- ◯ longer is meant to return a reference to a string, &String, but its parameters a and b automatically dereference the references passed in as arguments, and so are of type String.
- ◯ longer returns a reference to one of its two parameters, and since Rust is strongly typed, this requires that both parameters be constrained to have equivalent lifetimes.

10. Will the following compile? If not, what's an issue?

```rust
let mut x = String::from("Hello");
let y = &mut x;
let z = &x;
y.push_str(" World");
```

- ◯ It will compile.
- ◯ You are not allowed to create a mutable reference to mutable data, as this violates the "only one mutable reference" rule.
- ◯ You are not allowed to create an immutable reference to mutable data, as this violates the "only one mutable reference OR any number of immutable references" rule.
- ◯ Since z is an immutable reference, the mutable reference y cannot be used after z is created, as this violates the "only one mutable reference OR any number of immutable references" rule.

11.
```rust
struct Point {
  x: i32,
  y: i32,
}
impl Point {
  fn m(&mut self) {
    self.x += 1;
    self.y += 1;
  }
}
fn main(){
  let mut p = Point{ x: 0, y: 0 };
  p.m();
  println!("({}, {})", p.x, p.y);
}
```

What is printed?

- ○ `(1, 1)`
- ○ `(0, 0)`
- ○ There will be an error.

12.
```
enum Number {
  Zero,
  One,
  Two,
}

fn main() {
    use Number::Zero;
    let t = Number::One;
    match t {
        Zero => println!("0"),
        Number::One => println!("1"),
    }
}
```

    What is printed in this program?

    - ○ `0`
    - ○ 1
    - ○ 2
    - ○ Compile Error

13.
```
trait Trait {
   fn p(&self);
}

impl Trait for u32 {
   fn p(&self) { print!("1"); }
}

fn main(){
    let x=100;
    x.p();
}
```

    What is printed in this program?

    - ○ 100
    - ○ 1
    - ○ Runtime Error
    - ○ Compile Error

14.
```
fn main(){
  let mut x = String::from("Hello");                        (1)
  {
      let y = &x;                                            (2)
      println!("{} = {}",y,x);                               (3)
  }
  let z = &mut x;                                            (4)
  println!("({}, {})", z, x);                                 (5)
}
```

The above code doesn't compile and the error thrown by the rust compiler is:

```
error[E0502]: cannot borrow `x` as immutable because it is also
borrowed as mutable
 --> src/main.rs:8:27
  |
7 |   let z = &mut x;
  |           ------ mutable borrow occurs here
8 |   println!("({}, {})", z, x);
  |   ------------------------^-
  |   |                        |
  |   |                        immutable borrow occurs here
  |   mutable borrow later used here
```

Which of the following are some possible fixes:

14.1. Change line (4) to `let z = &x`
14.2. Change line (5) to `println!("{},{}",z,z);`
14.3. Remove line (3)
14.4. Change line (1) to `String::new` instead

- ◯ 1, 3
- ◯ 3, 4
- ◯ 1, 2
- ◯ 1
- ◯ 2
- ◯ 3
- ◯ 4
- ◯ 1, 2, 3
- ◯ All of the choices
- ◯ None of the choices

15.
```
enum Counter {
    Node(Option<Box<Counter>>)
}

fn main() {
    let mut my_list = Counter::Node(Some(Box::new(
        Counter::Node(Some(Box::new(
            Counter::Node(Some(Box::new(
                Counter::Node(Some(Box::new(
                    Counter::Node(None)
                )))
            )))
        )))
    )));

    loop {
        match my_list {
            Counter::Node(None) => {
                break;
            }
            Counter::Node(Some(b)) => {
                my_list = *b;
                println!("1, ");
            }
        }
    }
}
```

How many 1's does this print?

- ◯ 0
- ◯ 1
- ◯ 2
- ◯ 3
- ◯ 4
- ◯ 5
- ◯ 6
- ◯ 7
- ◯ 8

16.
```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    // Mark 1
    {let c = {
        let b = Rc::new(Cons(4, Rc::clone(&a)));
        // Mark 2
        let y = cons(7,&b);
        // Mark 3
        y
    };
    if let Cons(_,b) = c{
        // Mark 4
    }
    }
    // Mark 5
}

fn cons(y:i32,x:&Rc<List>)->List{
    Cons(y,Rc::clone(&x))
}
```

Use the above code to answer the following questions:

16.1. What is the count of `[5,10]` at Mark 1?
- ○ 0
- ○ 1
- ○ 2
- ○ 3
- ○ 4

16.2. What is the count of `[5, 10]` at Mark 2?
- ○ 0
- ○ 1
- ○ 2
- ○ 3
- ○ 4

16.3. What is the count of `[4, 5, 10]` at Mark 3?
- ○ 0
- ○ 1
- ○ 2
- ○ 3
- ○ 4

16.4. What is the count of `[4, 5, 10]` at Mark 4?
- ○ 0
- ○ 1
- ○ 2
- ○ 3
- ○ 4

16.5. What is the count of `[5, 10]` at Mark 5?
- ○ 0
- ○ 1
- ○ 2
- ○ 3
- ○ 4

16.6. What is the count of `[4, 5, 10]` at Mark 5?
- ○ 0
- ○ 1
- ○ 2
- ○ 3
- ○ 4

Answers

Q1. False
Q2. False
Q3. False
Q4. False
Q5.1. False
Q5.2. Hello == Hello
Q5.3. Hello World
Q6. Line 3 sets the value of z to 4, and so “4” is successfully printed.
Q7. Since y is a mutable reference, x cannot be used as a mutable reference again until the lifetime of y is over.
Q8. The lifetime of s ends at the end of make_vec, and so the pushed &s would become a dangling pointer once the vector is returned.
Q9. longer returns a reference to one of its two parameters, and since Rust is strongly typed, this requires that both parameters be constrained to have equivalent lifetimes.
Q10. Since z is an immutable reference, the mutable reference y cannot be used after z is created, as this violates the “only one mutable reference OR any number of immutable references” rule.
Q11. (1, 1)
Q12. Compile Error
Q13. 1
Q14. 1,2
Q15. 4
Q16.1. 1
Q16.2. 2
Q16.3. 2
Q16.4. 1
Q16.5. 1
Q16.6. 0