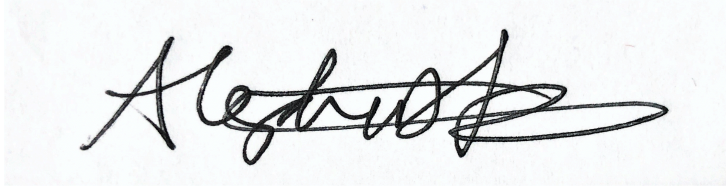


Name: Stephanie Doan
Student ID: 604981556

Time started exam: 12:10pm
Time ended exam: 1:39pm (took time to scan this signature)

A handwritten signature in black ink, appearing to read 'Stephanie Doan', written on a light-colored background.

Signature:

Date: 5/4/2020

QUESTION 1

A.

This call is erroneous because the return statement `print_endline "OK"` returns type `unit` while in the else statement, `"BAD"` returns simply type `string`. What is returned from the then statement and the else statement must be of the same type because a function must return values of a consistent type.

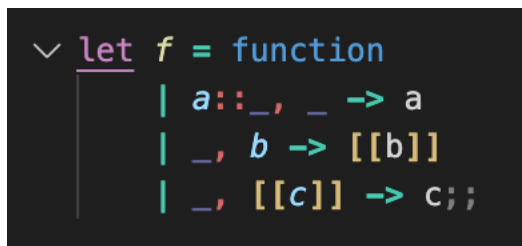
The extension to OCaml would be that there would be no binding of functions to a certain type, which would be checked at compile time before running the program. The downside of this is that there would be error at compile time when the types are inconsistent.

B.

Here, the second `"a"` is taken as the name of a function, as the first `"a"` in `"let a.."` makes `"a"` a function. OCaml tries to call the second `"a"` as this function but it cannot because there is no `"rec"` word describing function called `"a"`, so function `a` cannot be called recursively in this line.

The extension to OCaml would be to remove recursive and non-recursive binding

C.



```
✓ let f = function
  | a::_ , _ -> a
  | _ , b -> [[b]]
  | _ , [[c]] -> c;;
```

(retyped for readability)

The expected returned type in the second matching case is `[[b]]` in which `b` is of unknown type, but the double brackets make `b` 'a list list. In the third match case, `[[c]]` is equivalent to what `b` was matched to, so this would add two more layers of list nesting, making `[[b]]` a 'list list list list.

QUESTION 2

The first function definition is INVALID.

```
let rec a = fun x -> x a
```

Here, “a” is defined in the function statement “let rec a..” to be a function that returns a function that returns a value of unknown type, and this value is returned as the result of “a”. The type bound to “a” is ('a -> 'b) -> 'c. However, when x is called on a in “x a”, a is expected to be value of unknown type, since it takes no arguments and is simply called by function x. However, this is not the type bound to “a” already, so there is an error.

The second function can be called as such for example:

```
let x = 1;;
```

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

```
let rec b = fun x -> fun y -> y x;;
```

```
(b x y)
```

Here, y is a function called on one argument, and x simply returns an integer. This is an example function call that returns 1:

```
- : int = 1
```

QUESTION 3

```
let rec generous_matcher acceptors frag = match acceptors with
| [] -> None
| acceptors_head::acceptors_tail ->
    let res = acceptors_head frag in match res with
    | Some x -> x
    | _ -> generous_matcher acceptors_tail frag
```

```
let rec make_generous_matcher gram =
    match gram with (start_symbol, grammar) ->
        (fun acceptors frag -> generous_matcher acceptors frag) ;;
```

QUESTION 4

```
let revhalf L =  
  let rec f odd l =  
    if (List.length l) < 3  
    then l  
    else match l with head::middle::tail ->  
          if odd = 0  
          then tail::(f 1 middle)::head  
          else head::(f 0 middle)::tail  
  in  
  if (List.length L) % 2 = 1 then (f 1 L)  
  else match L with  
    | head::tail -> hd::(f tail 0)  
    | [] -> []
```

QUESTION 5

A. (I will rewrite only the rules that need re-writing to save time.)

Here I replace all occurrences of brackets and curly braces. Red lines are replaced with the black lines right after them.

Statement:

```
assert Expression [: Expression];  
assert Expression;  
assert Expression : Expression;
```

```
break [id];  
break;  
break id;
```

```
continue [id];  
continue;  
continue id;
```

```
return [Expression];  
return;  
return Expression;
```

```
try Block [Catches] Finally;  
try Block Finally;  
try Block Catches Finally;
```

Catches:

```
CatchClause {CatchClause};  
CatchClause;  
CatchClause Catches;
```

CatchType:

```
id { | id }  
id  
id | CatchType
```

Block:

```
{ BlockStatements }  
BlockStatements  
BlockStatements Block
```

BlockStatements:

```
{Statement}  
Statement  
Statement BlockStatements
```

B.

Yes it would work with Homework 2 solution because there is no left hand side recursion that would result in an infinite loop, meaning a rule that recurses into itself. Infinite recursion would break our parser because it cannot check for this.

C.

ParExpression:
Expression)

This wouldn't make the grammar ambiguous because tokens that are matched to ParExpression would still call the Expression rule in the same way. Removing the "(" token would not effect this chain.

D.

This would not produce ambiguity because we are simply replacing one token with another, so the parse trees produced would still be the same, though they would match with different strings.

E.

Removing finally would induce ambiguity because these two parse trees would make the same result.

Finally -> Block
Block

QUESTION 6

6a.

This wouldn't be correct because of a race condition. If thread A calls wait then sets the boolean flag to true, then immediately B calls notifyAll and sets flag to false, A doesn't reach the while loop yet. It sees flag is false then exits without waiting as it was supposed to.

6b.

This implementation is kind of like a spin lock, the thread that is waiting will run in an infinite loop waiting to acquire the lock, wasting CPU time where it could have given control to another thread that actually does something useful instead of running in an infinite loop that does nothing.

6c.

Synchronized would fix the race condition by locking the flag so that only one flag can change its value at a time. The problem of many spin locks would be alleviated because only one thread can enter that infinite loop waiting for access to the wait variable. Unlike HW3 where the synchronized keyword was placed around a critical section of code, here the synchronized is put around only the lock, essentially allowing the lock to behave the same but also preventing other locks from entering the infinite loops waiting, allowing them to perhaps do something useful.

QUESTION 7