Sample Midterm Solutions

UCLA CS 131 Mock LA Midterm, Winter 2025
100 minutes total, open book, open notes, closed computer
Write answers on exam.

Name: _____ Student ID: _____

1 (8 minutes). Your team has been tasked to write an application to become part of the company's access control system. The software will be exposed over the internet and must be secure and reliable - if it goes down you'll lock everyone out of the office! Give pros and cons of both OCaml and Java for developing this application.

Non-exhaustive solution ideas:

Pros for OCaml (resp. cons for Java) include:

- Richer type system that prevents certain errors (you can't misuse an option but you can misuse a reference that is possibly null).
- Immutability can help prevent data races (albeit not race conditions in general).
- More expressive code means it might be easier to audit the security-critical parts.

Pros for Java (resp. cons for OCaml) include:

- Mutability may improve performance - that can be useful if the system experiences heavy load or should run in real-time.
- Likely a larger and more battle-tested ecosystem of libraries for interacting with other systems like databases & the network.
- More widely used so there's more experts and advice on writing secure and reliable Java programs.

2. Suppose these types are defined in an OCaml program:

```
type nucleotide = A | C | T | G
type fragment = nucleotide list
```

Two nucleotides are *complementary* if one is A and the other T, or one is C and the other G.
Suppose there is a function `align` such that `(align frag1 frag2)` returns a tuple of:

- the number of positions where the nucleotides in the fragments at that position were not complimentary, and
- the "leftover" size; that is, the difference in length between `frag1` and `frag2`.
-]\;

2a (2 minutes). What is the type of the function align?

Solution: `nucleotide list -> nucleotide list -> int * int`

2b (10 minutes). Implement the function align. Keep it as syntactically simple as possible. You are allowed to use functions from OCaml's List module.

Solution:

```
type nucleotide = A | C | T | G
type fragment = nucleotide list

let complementary n1 n2 =
  match n1, n2 with
  | A, T | T, A | C, G | G, C -> true
  | _ -> false


let align frag1 frag2 =
  let rec helper non_compl lst1 lst2 = match lst1, lst2 with
    | [], [] -> (non_compl, 0)
    | _, [] -> (non_compl, List.length lst1)
    | [], _ -> (non_compl, List.length lst2)
    | h1::t1, h2::t2 ->
        if not (complementary h1 h2) then
          helper (non_compl + 1) t1 t2
        else
          helper non_compl t1 t2
```

```
in
helper 0 frag1 frag2
```

2c (4 minutes). Consider a situation where the align function is intended to work with a list of nucleotide types, but a lab programmer wants it to work with a list of RNA types (in addition to nucleotide types). She secretly adds the following line to the top of the existing code base:

```
type RNA = A | U | G | C
```

What is most likely to occur when the code is executed? Explain.

Solution:
The OCaml code would result in a semantic error due to the redefinition of the constructor A, G, and C. In OCaml, each constructor in a type definition must be unique within the entire module unless they are scoped differently, for example, by using modules or polymorphic variants. Since A, G, and C are defined both in type nucleotide and type RNA, this will cause a conflict when type matching happens in the complementary function as it types its input fragments as RNA and thus fails to pattern match with T.

2d (4 minutes) Provide an example of OCaml code that demonstrates how to safely access the head of a list using the Option type to handle cases where the list may be empty. Explain why using the Option type is helpful in this context.

Solution:

```
let head = function
  | [] -> None
  | x :: _ -> Some x
```

Benefits:

- Handling Absence Explicitly: In many programming contexts, values may be optional, meaning they might exist or not. The option type (Some or None) provides a safe way to handle such scenarios, ensuring that the program always accounts for the case where a value might be missing.
- Avoiding Null Errors: In languages that use null references to represent missing values, null pointer exceptions are common and

often a source of bugs. OCaml's Option type avoids this problem by replacing nullable references with a type-safe alternative.

- Improving Code Clarity.

3 (20 minutes total). Consider the following EBNF grammar for a match expression in the fictional language Fe2O3. Items in [brackets] are optional. Items in {curly braces} can be repeated zero or more times; if the closing brace is followed by '+' the item must occur at least once. '|' separates alternatives. Terminal symbols are "quoted". Some terminals, such as 'INTEGER', are meant to represent, e.g. integer Literals instead of the characters 'INTEGER'. The start symbol is 'MatchExpression'. (* Comments look like this *)

```
MatchExpression ::= "match" Scrutinee "{" [MatchArms] "}"

Scrutinee ::= Expression

MatchArms ::= {MatchArm "=>" Expression ","}
              MatchArm "=>" Expression [","]

MatchArm ::= Pattern [MatchArmGuard]

MatchArmGuard ::= "if" Expression

Expression ::= "INTEGER"                    (* Integer literal *)
             | "VARNAME"                    (* Variable name *)
             | Expression {"," Expression}+ (* Tuples *)
             | "(" Expression ")"

Pattern ::= "INTEGER"               (* Integer pattern *)
          | "VARNAME"               (* Capture pattern *)
          | "_"                     (* Wildcard pattern *)
          | Pattern {"," Pattern}+  (* Tuple patterns *)
          | "(" Pattern ")"
```

3a (2 minutes). Are there any blind-alley rules in this grammar? If yes, please identify them. If not, could you add one?

Solution: No.

There are many ways to add a blind-alley rule. Here is one - invent a nonterminal and add this rule:
BlindMatchArm ::= Pattern BlindMatchArm

3b (10 minutes). Convert the grammar to BNF Form, resolving precedence-related ambiguities the same way that OCaml resolves them (i.e. precedence rules are the same as in OCaml). Assume any blind alley rules are removed.

Solution:

```
MatchExpression ::= "match" Scrutinee "{" MatchArmsOpt "}"

Scrutinee ::= Expression

MatchArmsOpt ::= MatchArms
               | ""

MatchArms ::= Arrow "," MatchArms
            | Arrow ","
            | Arrow

Arrow ::= MatchArm "=>" Expression

MatchArm ::= Pattern MatchArmGuard
           | Pattern

MatchArmGuard ::= "if" Expression

(* Expression/Expression1 implements precedence *)

Expression ::= Expression "," Expression1
             | Expression1

Expression1 ::= "INTEGER"
              | "VARNAME"
              | "(" Expression ")"

Pattern ::= Pattern "," Pattern1
          | Pattern1

Pattern1 ::= "INTEGER"
           | "VARNAME"
           | "_"
           | "(" Pattern ")"
```

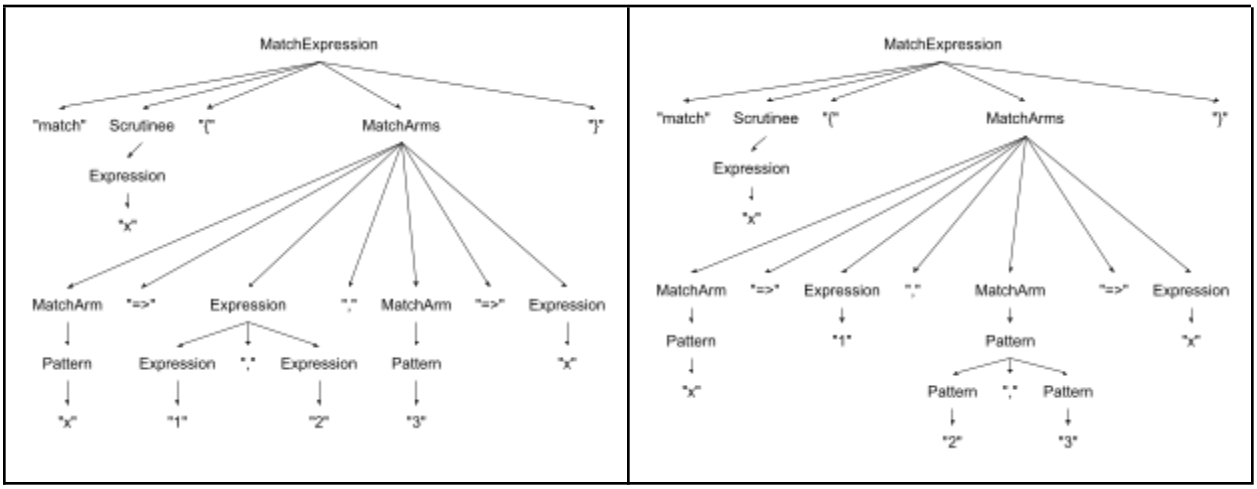
3c (8 minutes). We are considering removing the delimiting commas in MatchArms (highlighted in yellow). Will this make the grammar more or less ambiguous? Give an example of any ambiguities that the change

introduces, or any ambiguities in the original grammar that the change resolves.

This change resolves an ambiguity in original grammar where the Expression in one match arm runs into the Pattern in the next match arm. Consider the string

```
match x { x => 1 , 2 , 3 => x }
```

There are two possible parse trees in the original grammar:



4 (10 minutes total).

4a (4 minutes). Given two pointer types in C: “int *a” and “const int *b”, can we assign “b = a” without an explicit cast? Explain why.

Solution:
Recall that const int * means that the integer being pointed to can't be modified, so anything we can do with a const int * (read), we can do with an int * (read, write). According to Liskov's substitution rule, int * is a subtype of const int *.

```
#include <stdio.h>

int main() {
    const int value = 10;
    const int *a = &value;      // a is a pointer to a const int
    int *b;                     // b is a pointer to an int

    b = a; // Compilation error: discards 'const' qualifier
    a = b; // This line causes no compilation error
}
```

4b (6 minutes). Consider the following Java code snippet:

```
public class DetailedExample {

    public void display(Integer number) {
        System.out.println("Integer version: " + number);
    }
    public void display(Double number) {
        System.out.println("Double version: " + number);
    }
    public void display(Object obj) {
        System.out.println("Object version: " + obj.toString());
    }

    public static void main(String[] args) {
        DetailedExample example = new DetailedExample();
        example.display(500);    // ....1
        example.display(100L);   // ....2
        example.display(5.0);    // ....3
```

```
            example.display(null);   // ....4
        }
    }
```

Will this code run successfully? If so, then what will it output? If not, what can you change to prevent a compilation error? Explain.

Solution:

- Line 4 will cause an error due to ambiguity in polymorphism: Both, display(Integer number) & display(Double number) match example.display(null); type coercion only works when there is no polymorphic ambiguity.
- To fix, delete either one of the methods causing the ambiguity: display(Integer number) or display(Double number).

5 (8 minutes total). What are the types of the following OCaml expressions?

5a (2 minutes).

```
(* Function applying a given function twice to an argument *)
let apply_twice f x = f (f x)
```

Solution: ('a -> 'a) -> 'a -> 'a
The reason there is only 'a and not 'a/'b is because the return value of f is then given as a parameter to f. So the return type and parameter type must be the same.

5b (2 minutes).

```
(* Function to generate curried multipliers *)
let dual_multiplier n m = fun x -> n * m * x
```

Solution: int -> int -> int -> int

5c (2 minutes).

```
(* Extend map_pair to map_triple *)
let map_triple f g h (x, y, z) = (f x, g y, h z)
```

Solution:
('a -> 'b) -> ('c -> 'd) -> ('e -> 'f) -> 'a * 'c * 'e -> 'b * 'd * 'f

5d (2 minutes).

```
(* A curried function generator that dynamically creates a function
chain based on an initial seed *)
let dynamic_chain seed =
  let rec build_chain n f =
    if n = 0 then f
    else build_chain (n - 1) (fun x -> apply_twice f x)
  in
  build_chain seed
```

Solution: int -> ('a -> 'a) -> 'a -> 'a

6 (12 minutes total). Consider a grammar designed to represent well-formed parentheses strings. A string of parentheses is well-formed if each opening parenthesis "(" is matched with a closing parenthesis ")" in the correct order. The grammar rules for such a language can be informally described as follows:

S -> SS
S -> (S)
S -> ε (where 'ε' represents the empty string)

This grammar generates strings like "", "()", "(())", "()(())", etc., all of which are examples of well-formed parentheses. For recursive and syntactical simplicity, we will represent a string using a list of characters.

6a (10 minutes). Write an OCaml function is_well_formed that takes a list of characters representing a sequence of parentheses and returns 'true' if the list of characters/string is well-formed according to the above grammar, and 'false' otherwise.

Example Output:
is_well_formed [] should return true
is_well_formed ['(' ; ')'] should return true
is_well_formed ['(' ; '(' ; ')' ; '(' ; ')' ; ')'] should return true
is_well_formed ['(' ; ')' ; '('] or is_well_formed [')' ; '(']
should return false

Solution:

```
let is_well_formed lst =
  let rec helper lst depth =
    match lst with
    | [] -> depth = 0  (* Return True if all opened parentheses are closed! *)
    | '(' :: t -> helper t (depth + 1)  (* Increment depth for each '(' *)
    | ')' :: t -> if depth > 0 then helper t (depth - 1) else false
       (* Decrement depth for each ')' if depth is positive *)
    | _ -> false (* Invalid character*)
  in
  helper lst 0;;
```

6b (2 minutes). What is the type of is_well_formed?

Solution: char list -> bool

7 (4 minutes). Suppose that an exception is thrown from a synchronized method while the method's object is being accessed by another thread. What should the Java Memory Model do in this situation? Think about exception handling and how that can be done gracefully.

If a single method in one thread throws an exception, it needs to be handled gracefully. The entire program cannot crash. Use a try-catch statement that allows the thread to return and release the lock on the object. Then, other threads can see the result from the failed thread and potentially deal with it.

8 (8 minutes). Suppose you are developing a multithreaded application in Java. We are in full control of our application's threads, and we know beforehand which threads' computations need to be prioritized and which threads will be IO-bound, so we want to cooperatively schedule our threads. Is it possible to do this by just using Java's Thread.yield() method? If so, describe how you would do it. Otherwise, explain why not, and describe what additional components would be needed to achieve cooperative multithreading.

Cooperative multithreading with just yield is not possible since Java Threads are scheduled by the OS, which we have no control over at the language level. We would need to implement our own scheduler, and our own user threads to achieve cooperative scheduling. For example, we could have an event loop that processes each thread of execution for a bounded amount of time, and switches to a different thread depending on the priority. In order to take advantage of parallelism, we'll still need to use multiple OS threads to utilize multiple CPUs, but there will need to be a layer of abstraction so that user threads do not correspond one-to-one to OS threads.

9 (10 minutes). Compare and contrast Java interfaces with OCaml variants. When would you use one over the other? Consider performance, simplicity, and extensibility.

Similarities

- Abstraction over different data types
- Similar performance - interfaces require some sort of vtable, while variants are a tagged union

When you would use interfaces

- More extensible, especially if you are writing a library. Library users can simply implement your interface on their classes. To extend a variant, you would have to modify the variant definition and add cases everywhere the variant is checked.

When you would use variants

- When the data type you are modeling has finite variants, e.g. nodes in a tree can only be leaf or parent nodes, or an option can only be Some or None.
- You can use pattern matching to distinguish variants, but can't distinguish between different objects implementing the same interface