UNIVERSITY OF RUHUNA

Faculty of Engineering

End-Semester 8, Examination, January 2026

Module No: EC8206 **Module Name: Functional Programming**

Part-A

[One Hour]

Instructions for candidates

- You must attempt Part A first.
- Part A will be collected after one hour.
- Answer in the Answer Sheet Provided.
- Question paper contains 40 multiple choice questions.
- Answer all questions. Each question has only one answer.
- **For each question, put an X mark on the letter: (a), (b), (c), or (d) which corresponds to the correct answer, by using a black or blue pen.**
- Each correct answer caries 0.5 marks.
- Once Part A is submitted, you may proceed to Part B, which has a time allocation of 2 hours.

---

1. What is a typeclass in Haskell?
   a. A way to define a new type of data
   b. A way to define a set of operations that can be performed on a type of data
   c. A way to define a set of functions that operate on a type of data
   d. A way to define a set of constraints on a type of data
2. What is the difference between a list and a tuple in Haskell?
   a. A list is mutable and a tuple is immutable
   b. A list is a sequence of elements of the same type, while a tuple can contain elements of different types
   c. A list can have a variable length, while a tuple has a fixed length
   d. A list can be used for pattern matching, while a tuple cannot
3. Which of the following is the correct way to define a function with a default parameter value in Haskell?
   a. f x = x + 1 (default 0)
   b. f x y = x + y (default 1)
   c. f x y z = x + y + z (default 0)
   d. Haskell does not support default parameter values

4. What's the difference between Int and Integer?

   a. Int is arbitrary precision, Integer is fixed-width
   b. Integer is arbitrary precision, Int is machine-dependent
   c. Int supports decimals, Integer doesn't
   d. They are identical

5. Which type class provides the (==) operator?

   a. Ord
   b. Num
   c. Eq
   d. Show

6. Which type class constraint is needed for elem :: a -> [a] -> Bool?

   a. Num a
   b. Eq a
   c. Ord a
   d. Show a

7. What advantage does recursion provide over iteration?

   a. Natural expression of recursive problems
   b. No need for mutable state
   c. Easier mathematical reasoning
   d. All of the above

8. What's wrong with this function?

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

   a. Missing type signature
   b. Non-exhaustive patterns for negative numbers
   c. Should use guards instead
   d. Nothing is wrong

9. Convert to guarded equation:

```
classify n
| n < 0 = "negative"
| n == 0 = "zero"
| otherwise = "positive"
```

   a. Uses pattern matching
   b. Uses lambda expressions
   c. Uses conditional branching
   d. Uses case expressions

10. What's the issue here?

```
f [] = []
f (x:xs) = x : f xs
```

    a. Missing base case
    b. Will cause stack overflow
    c. No issue - it's the identity function for lists
    d. Should use ++ instead of :

11. What does

$$[x^2 \mid x \leftarrow [1..5], \text{odd } x]$$

    evaluate to?

    a. [1,9,25]
    b. [1,4,9,16,25]
    c. [1,9]
    d. [1,3,5]

12. What does zipWith (+) [1,2,3] [4,5,6] produce?

    a. [(1,4), (2,5), (3,6)]
    b. [5,7,9]
    c. [1,2,3,4,5,6]
    d. Type error

13. Convert this comprehension to higher-order functions:

```
[ toUpper c | c <- "hello", c `elem` "aeiou"]
```

    a. map toUpper (filter (elem' "aeiou") "hello")
    b. filter (`elem` "aeiou") (map toUpper "hello")
    c. foldr (acc -> toUpper c : acc) [] "hello"
    d. zipWith toUpper "hello" "aeiou"'

14. Why is recursion fundamental in FP?

    a. It's faster than iteration
    b. Immutability makes iteration impossible
    c. It's the only way to handle lists
    d. It enables formal verification

15. What does this recursive function do?

```
mystery _ [] = []
mystery f (x:xs) = f x : mystery f xs
```

    a. filter
    b. map

c. foldr

d. zipWith

16. Tail-recursive factorial:

```
fact n = go n 1
where go 0 acc = acc
go n acc = go (n-1) (n*acc)
```

Why is this better?

a. Uses less memory

b. Prevents stack overflow

c. Both a and b

d. It's not better

17. What's the type of this multiple argument recursion:

```
power _ 0 = 1
power x n = x * power x (n-1)
```

a. Num a => a -> a -> a

b. Int -> Int -> Int

c. a -> Int -> a

d. Num a => a -> Int -> a

18. Why are Higher-Order Functions useful?

a. They enable function composition and abstraction

b. They make code faster

c. They're required for type classes

d. They enable mutation

19. What's the difference between map and fmap?

a. fmap can handle multiple arguments

b. fmap works for any Functor, map only for lists

c. map is faster

d. They're identical

20. What is pattern matching in Haskell?

a. A way to match patterns of expressions to specific data

b. A way to match patterns of functions to specific expressions

c. A way to match patterns of data to specific expressions

d. A way to match patterns of expressions to specific functions

21. What does foldr (:) [] do?

a. Reverses a list

b. Identity for lists
c. Sums a list
d. Filters a list

22. What does this demonstrate?

```
(f . g) x = f (g x)
```

a. Partial application
b. Currying
c. Lambda calculus
d. Function composition

23. Why does foldl cause space leaks?

a. It uses too much stack
b. It builds thunks instead of evaluating
c. It's not tail-recursive
d. It copies the list

24. What's the result of:

```
let f = foldr (&&) True in f [True, False, True]
```

a. True
b. False
c. [True, False, True]
d. Error

25. What does this function do?

```
f [] = True
f [_] = True
f (x:y:zs) = (x <= y) && f (y:zs)
```

a. Sorts the list
b. Checks if list is sorted
c. Removes duplicates
d. Reverses the list

26. Why does immutability help with parallelism?

a. Better cache performance
b. No locks needed for shared data
c. Automatic parallelization
d. Both a and b

27. What does this test?

```
prop_reverse :: [Int] -> Bool
prop_reverse xs = reverse (reverse xs) == xs
```

a. reverse is its own inverse
b. reverse is commutative
c. reverse is associative
d. reverse is identity

28. What is currying in Haskell?

a. A technique for transforming a function that returns a value into a function that does not return a value
b. A technique for transforming a function that takes a single argument into a function that takes multiple arguments
c. A technique for transforming a function that takes no arguments into a function that takes one or more arguments
d. A technique for transforming a function that takes multiple arguments into a series of functions that each take a single argument

29. Which of the following is not a built-in data type in Haskell?

a. Integer
b. Char
c. Boolean
d. Double

30. What is a partial function in Haskell?

a. A function that is only defined for some values of its input
b. A function that is defined for all values of its input
c. A function that generates random values
d. A function that transforms one type of data into another

31. Which of the following is the correct way to define a recursive function that calculates the factorial of a number in Haskell?

a. factorial n = if n == 0 then 1 else n * factorial n-1
b. factorial n = if n == 0 then 1 else n * factorial (n-1)
c. factorial n = if n == 1 then 1 else n * factorial n-1
d. factorial n = if n == 1 then 1 else n * factorial (n-1)

32. What does this list comprehension produce?

```
[x * 2 | x <- [1..5], odd x]
```

a. [2,6,10]
b. [1,2,3,4,5]

c. [2,4,6,8,10]
d. [2,4,6,8,10]

33. What does this dependent generator create?

```
[(i,j) | i <- [1..3], j <- [1..i]]
```

a. [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
b. [(1,1),(2,2),(3,3)]
c. [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
d. [(1,1),(2,2),(3,3)]

34. What does this nested comprehension produce?

```
[[x * y | y <- [1..3]] | x <- [1..2]]
```

a. [[1,2,3], [2,4,6]]
b. [1,2,3,2,4,6]
c. [[1,2], [2,4], [3,6]]
d. [1,2,2,4,3,6]

35. What does this nested map produce?

```
map (map (+1)) [[1,2],[3,4]]
```

a. [[1,2,3,4]]
b. [2,3,4,5]
c. [[2,3], [4,5]]
d. [[3,7]]

36. What is equivalent to map f (map g xs)?

a. map f . map g xs
b. map g . map f xs
c. map (g . f) xs
d. map (f . g) xs

37. What does this filter with composition do?

```
filter ((==0) . (`mod` 3)) [1..10]
```

a. [3,6,9]
b. [1,4,7,10]
c. [2,5,8]
d. [1,2,4,5,7,8,10]

38. Convert this comprehension to map/filter:

```
[x*2 | x <- [1..5], x*2 > 5]
```

a. map (*2) (filter (-> x*2 > 5) [1..5])
b. filter (>5) (map (*2) [1..5])
c. map (*2) [1..5]
d. filter (-> x*2 > 5) [1..5]

39. What distinguishes concurrency from parallelism in Haskell?

a. Concurrency requires par, parallelism uses forkIO
b. Concurrency is about structure, parallelism is about execution
c. Only parallelism works with pure functions
d. They are synonymous in Haskell

40. What is wrong with this pattern matching?

```
sumFirstTwo [] = 0

sumFirstTwo [x] = x

sumFirstTwo (x:y) = x + y
```

a. Missing cases
b. Should use guards instead
c. Can't add x + y where y is a list
d. Nothing is wrong