Office Use Only

MONASH University

Semester Two 2018
Examination Period

Faculty of Information Technology

EXAM CODES: **FIT2102**

TITLE OF PAPER: **Programming Paradigms – Paper 1**

EXAM DURATION: 2 hours writing time

READING TIME: 10 minutes

THIS PAPER IS FOR STUDENTS STUDYING AT: (tick where applicable)

☐ Berwick	☑ Clayton	☑ Malaysia	☐ Off Campus Learning	☐ Open Learning
☐ Caulfield	☐ Gippsland	☐ Peninsula	☐ Monash Extension	☐ Sth Africa
☐ Parkville	☐ Other (specify)			

During an exam, you must not have in your possession any item/material that has not been authorised for your exam. This includes books, notes, paper, electronic device/s, mobile phone, smart watch/device, calculator, pencil case, or writing on any part of your body. Any authorised items are listed below. Items/materials on your desk, chair, in your clothing or otherwise on your person will be deemed to be in your possession.

No examination materials are to be removed from the room. This includes retaining, copying, memorising or noting down content of exam material for personal use or to share with any other person by any means following your exam.
Failure to comply with the above instructions, or attempting to cheat or cheating in an exam is a discipline offence under Part 7 of the Monash University (Council) Regulations, or a breach of instructions under Part 3 of the Monash University (Academic Board) Regulations.

AUTHORISED MATERIALS

OPEN BOOK	☐ **YES**	☑ **NO**
CALCULATORS	☐ **YES**	☑ **NO**
SPECIFICALLY PERMITTED ITEMS **if yes, items permitted are:**	☐ **YES**	☑ **NO**

Candidates must complete this section if required to write answers within this paper

STUDENT ID: __ __ __ __ __ __ __ __ DESK NUMBER: __ __ __ __ __

INSTRUCTIONS TO CANDIDATES:

- This paper contains ten (10) multiple-choice questions in Section A, and ten (10) short answer and coding questions in Section B. All questions in both sections should be attempted.

- **For the multiple choice questions** circle the most correct answer to each question in this exam booklet. Clearly circle only one answer for each question. If you make a mistake, cross it out and circle the correct answer.

- The examination is out of 100 marks, which contribute 40% towards your final assessment. Marks for individual sections and questions are clearly indicated throughout the exam paper.

- For the short answer and coding questions in Section B, you may answer the questions in any order. Boldly number your answers to all questions. Commence all sections on a new page.

- Write your answers to questions from Section B on the lined pages of the exam script. You may use the blank sides of the page for workings, however these pages will not be marked.

- State any assumptions that you make regarding any question.

- The examination end is marked with "END OF EXAMINATION"

Section A: Multiple Choice Questions
(40 marks total, 4 marks each)

Question 1. The Lambda Calculus is **NOT**:

A) A model for computation upon which functional programming is based.
B) A formal system where function application is left associative.
C) Capable of computing anything that a Turing Machine can compute.
D) Able to perform loops through functions which can refer to themselves.

Question 2. Which of the following statements about *closures* is **false**:

A) A closure has access to its enclosing function's variables even after the enclosing function has returned.
B) A closure can be used to store data.
C) A closure cannot access its enclosing function's parameters.
D) The variable references inside the closure are not evaluated until the closure is called.

Question 3. Consider the following TypeScript code:

```
const a = [1,2,3]
```

Which of the following will **not** cause a compile error:

```
A) a[2] = 4
B) a = [2,3,4]
C) a[1] = "3"
D) const b:Array<string> = a
```

Question 4. Consider the following JavaScript code:

```
const g = f => h=> x=> f(h(x))
```

Which statement most correctly describes `g`:

Why isn't A acceptable?

A) `g` holds the value returned by `f(h(x))`
B) `g` is a closure which returns a function called `f` that will be applied to `h` and `x`
C) `g` is the composition of unary functions f and h
D) `g` will cause an exception at runtime because `f` is not a function

Question 5. Consider the following code using the Observable definition we discussed in lectures and in the assignment:

```
const o = Observable
        .fromEvent<MouseEvent>(document, "mousemove")
        .map(({clientX, clientY})=> ({x: clientX, y: clientY})),
  u1 = o.map(({x,y}) => `${x},${y}`).subscribe(s => p.innerHTML = s),
  u2 = o.filter(({x}) => x > 400)
       .subscribe(_ => p.classList.add('highlight')),
  u3 = o.filter(({x}) => x <= 400)
       .subscribe(_ => p.classList.remove('highlight'));
```

Which of the following statements about this code is **false:**

A) The functions passed to map are pure.

B) Only one mousemove event listener will be added to the document element

C) The functions passed to subscribe are not pure

D) u1, u2 and u3 are functions which should be called when the behaviour set up by this code is no longer required.

Question 6. Consider the following Haskell function definition:

```
test :: Eq a => a -> a -> Bool
test a b = a == b
```

What is the type of the following expression?

```
test 1
```

A) `Bool`

B) `(Eq a, Num a) => a -> Bool`

C) `Num a => a -> a -> Ordering`

D) `(Num a, Eq b) => a -> b -> Bool`

Question 7. Recall that in Haskell foldl has type:

```
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

What is the result of the following expression?

```
foldl (-) 0 [1..10]
```

(Hint: don't assume anything – look carefully at the type of foldl)

A) 5

B) 0

C) -55

D) -5

Question 8. What type will the Haskell compiler infer for the following function?

```
lift f a b = f <$> a <*> b
```

Eg: lift (+) [3, 4] [1, 2]
= [(+3), (+4)] <*> [1, 2]
= [4,5,5,6]
"[] is a container f"

A) `lift :: Monad f => f a -> f b -> (a -> b) -> f b`
B) `lift :: Applicative f => (a1 -> a -> b) -> f a1 -> f a -> f b`
C) `lift :: Functor f => f (a1 -> b) -> a -> [a1] -> [b]`
D) `lift :: Traversable f => (a1 -> a -> b) -> f a1 -> f a -> f b`

Eg: lift (+) (Just 5) (Just 3)
= Just (+5) <*> Just 3
= Just 8
"Just is a container f"

Question 9. Which one of the following Lambda Calculus expressions is **not** divergent?

expression to be reduced forever

A) (λz . z z) (λy . y y)
B) λf. (x. f(x x)) (x. f(x x))
C) (λx. x x) (λx. x x)
D) (λx. x x) y

Question 10. Which of the following MiniZinc constraints does not imply that all variables in the array x take different values?

A) `forall (i in 1..n-1) (x[i] != x[i+1])`
B) `forall (i in 1..n, j in 1..n where i<j) (x[i] < x[j])`
C) `forall (i in 1..n, j in i+1..n) (x[i] != x[j])`
D) `forall (i in 1..n-1) (x[i] < x[i+1])`

b and c ensures all different because of nested loop which checks all combinations
d ensures all different because of transitivity - a < b and b < c, so a < c, i.e., list in increasing order

Section B: Short Answer and Coding Questions (60 marks total)

Question 11. (2 marks) Rewrite the following function in curried form in TypeScript. Include type annotations. **Use the same syntax for functions used here (i.e. do not use arrow syntax)**

```
function divide(a, b) {
  return a / b;
}
```

```
function divide(a:number) {
  return function (b:number) {
    return a/b;
  }
}
```

(2 marks) Write the function again using **arrow syntax**. Include type annotations.

```
const divide = (a:number) => (b:number) => a/b
```

Question 12. (6 marks) Write a TypeScript function that takes an uncurried binary function and returns a curried version.
Either long form function or arrow syntax or a mix is fine for this. Give type annotations such that the function is as generic as possible.

```
function curry<T,U,V>(f: (a:T,b:U) => V): ((a:T)=>(b:U)=>V) {
  return a => b => f(a,b);
}
```

Question 13. (8 marks total) This question is in three parts. All code must be self-contained, do not use any functions from the Prelude or other libraries.

(i) **(2 marks)** Write a **non-tail recursive** Haskell function with the following type:

```
sumZeroTo :: Integer -> Integer
```

It should return the sum of all numbers between (and including) itself and 0. It should also work for negative numbers. Thus:

```
> sumZeroTo 3
6
> sumZeroTo (-3)
-6
```

```
sumZeroTo 0 = 0
sumZeroTo n = n + (sumZeroTo (if n < 0 then n+1 else n-1))
```

(ii) **(3 marks)** Now write the function again **using tail recursion**. It should still expose the same type definition as above.

```
sumZeroTo n = s n 0
```

```
where s 0 a = a
      s n a = s (if n < 0 then n+1 else n-1) (n+a)
```

(iii) **(3 marks)** Which of the above implementations is better, and why?

The tail recursive version can be optimised to use constant memory by the compiler, while the non-tail recursive version could overflow the stack.

Question 14. (6 marks) Given the following Lambda Calculus expressions:

ONE = λf x. f x
NEXT = λn f x. f (n f x)

If TWO = NEXT ONE, what is the Lambda Calculus expression for TWO?

Give all of the reduction steps in your evaluation using the notation for substitution described in the lectures, i.e. `[<variable> := <substitute expression>].`

TWO = NEXT ONE
= (λn f x. f (n f x)) (λf x. f x)
= (λn f x. f (n f x)) (λg y. g y) // alpha conversion to avoid confusion
= λn[n:= (λg y. g y)] f x. f (n f x) // argument substitution
= λf x. f ((λg y. g y) f x) // beta reduction
= λf x. f ((λg[g:=f] y[y:=x]. g y) f x) // substitution
= λf x. f (f x) // beta reduction

TWO = λf x. f (f x)

Question 15. (6 marks total) Consider the following JavaScript definitions:

```
const
   cons = x=> y=> f=> f(x)(y),
   head = l=> l(K),
   tail = l=> l(K(I));
const aList = cons(1)(cons(2)(cons(3)(undefined)));
```

```
output:
> head(aList)
1
> head(tail(tail(aList)))
3
```

(i) **(2 marks)** Give JavaScript (i.e. no type annotations) for `K` and `I`.

```
K = x => y => x
I = x => x
```

(ii) **(4 marks)** Consider the following definition:

```
const fold = f=> i=> l=> l ? fold(f)(f(i)(head(l)))(tail(l)) : i
```

Give a JavaScript implementation of `forEach` using **both** `fold` and `K`, such that:

```
> forEach(console.log)(aList)
1
2
3
```

```
forEach = f=>l=>fold(K(f))(undefined)(l)
```

Question 16. (6 marks total) Implement the following function:

```
sequence :: Applicative f => [f a] -> f [a]
```

use only: `<$>, <*>, (:), foldr, pure`

```
sequence = foldr (lift (:)) (pure [])
             where lift f a b = f <$> a <*> b
```

Question 17. (6 marks)

Rewrite the following function in point-free form. On separate lines, show (and name) all of the reduction and conversion steps required to achieve the point-free form.

```
f a b = sqrt (a / b)
```

```
f a b = sqrt ((a/) b)  -- operator section
```

```
f a = sqrt . (a/)      -- function composition with (.)
f a = sqrt . ((/) a)   -- prefix form of operator
f a = (sqrt .) ((/) a) -- make precedence explicit with brackets
f = (sqrt .) . (/)     -- function composition with (.)
```

Question 18. (6 marks) A *multiplicative magic square* is a *n* x *n* matrix filled with the numbers 1, …, $2^{n \times n}$ such that the product of the numbers in each row, column and diagonal is equal. We call this product the *multiplicative magic number*. The following is a template MiniZinc model that contains the parameter *n*, variable declarations *x* (for the matrix) and *y* (for the multiplicative magic number), and the constraints for the diagonals. Add the two missing constraints that enforce that *x* is a magic square with magic number *y*.

```
include "alldifferent.mzn";
int: n;
array[1..n, 1..n] of var 1..2^(n*n): x;
var int: y;
constraint alldifferent(x);
constraint product (i in 1..n) (x[n-i+1,i]) = y;
constraint product (i in 1..n) (x[i,i]) = y;
```

```
Solution:
constraint forall (i in 1..n) (product(x[i,..]) = y);
constraint forall (i in 1..n) (product(x[..,i]) = y);

or (without array slicing):
constraint forall (i in 1..n) (product (j in 1..n) (x[i,j]) = y);
constraint forall (i in 1..n) (product (j in 1..n) (x[j,i]) = y);
```

Question 19. (6 marks) Recall the type definition of the flatMap function for `Observable`:

```
class Observable<U> {
  flatMap<V>(streamCreator: (_: U) => Observable<V>): Observable<V>
}
```

(i) **(3 marks)** Assuming we have a type class: `Observable a` in Haskell, where `a` is the type of elements being produced by the `Observable`, give the type definition for an equivalent `flatMap` function in Haskell (you only need to give the type definition, not the implementation).

flatMap :: Observable a -> (a -> Observable b) -> Observable b

```
flatMap :: Observable f => f a -> (a -> f b) -> f b
```

(ii) **(1 marks)** What type class from the Haskell Prelude has a function with an equivalent type?

```
Monad type class
```

(iii) **(2 marks)** Name that function and write its Haskell type definition.

```
Equivalent to type of bind function:

(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Question 20. (6 marks total)

We want to create a parser for a calculator able to compute addition and multiplication (only) that can take as input simple mathematical expressions over positive integers of the form:

12 + 21 * 3
6 * 4 + 333 + 8 * 24

(3 marks) Give algebraic data types with constructors `Number`, `Times`, `Plus`, which can be used to model the parse trees for such expressions.
`Number` should hold values as `Integer`.

```
data Expr = Number Integer | Times Expr Expr | Plus Expr Expr
```

(3 marks) Write a Haskell function to traverse the tree, evaluate the expressions and return the `Integer` result.

```
eval :: Expr -> Integer
eval (Number x) = x
eval (Times l r) = eval l * eval r
eval (Plus l r) = eval l + eval r
eval (Minus l r) = eval l - eval r
```

END OF EXAMINATION