Office Use Only

MONASH University

Semester Two 2018
Examination Period

Faculty of Information Technology

EXAM CODES:	**FIT2102**
TITLE OF PAPER:	**Programming Paradigms – PRACTICE EXAM**
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)			

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)

NOTE: For the purpose of the practice exam I am not writing these as multiple choice because I want to prompt you to ensure you really understand the concepts rather than some bare minimum. These questions are of equivalent difficulty and cover the topics of Section A of the real exam.

Question 1. Compare and relate the von Neumann architecture, Turing machines and Lambda Calculus. What are they? What programming paradigms have they influenced?

Lecture 1 and Lecture 5. von Neumann influenced modern computer architecture and arguably imperative programming in Assembly language. Turing machines are a theoretical construct but are also very imperative in terms of spreading their state all over the tape. Lambda calculus was a mathematical model for function application Alonzo Church that is the foundation and inspiration of functional programming.

Question 2. Precisely what data does a closure have access to and how long does it retain access to that data?
variables that are within scope for the enclosing function
the parameters to the closure
if the closure is called outside the scope the function from which it is called, it will still have access to any variables it references from the enclosing scope as well as its parameters.

Question 3. When a variable is declared const in JavaScript is it mutable or immutable? Is the object that it references mutable or immutable?
Lecture 2:
The variable is immutable, the object it references is mutable (unless caused to be immutable by various trickery that we don't care about for the purposes of this unit).

Question 4. In lectures we constructed lists of closures in JavaScript (which we called "cons lists").

```
const cons = x=> y=> f=> f(x)(y)
```

What exactly are the symbols cons, x, y, and f in the above code? Use words like function, parameter, argument, closure, apply/applied/application and so on.
cons is an immutable variable whose value is a function which takes a single parameter *x*.
when invoked with an argument, cons returns another function whose parameter is *y*, and which includes *x in its closure.*
This function in turn returns a unary function which expects a function *f* as argument and holds both x and y in its closure. When invoked this last function applies f to the values in x and y.

Question 5. What do the following terms mean and how do they relate to one another?

referential transparency

An expression is said to be referentially transparent if it can be replaced with its corresponding value without changing the program's behaviour. As a result, evaluating a referentially transparent function gives the same value for same arguments. Such functions are called pure functions. An expression that is not referentially transparent is called referentially opaque.

pure function

The function always evaluates the same result value given the same argument value(s) (so such a function has the property of being referentially transparent). The function result value cannot depend on any hidden information or state that may change while program execution proceeds or between different executions of the program. Neither can the function cause any (side-) effects (changes of state outside the scope of the function) beyond the construction of the return value.

declarative programming

A style of building the structure and elements of computer programs that expresses the logic of a computation without describing its control flow

mutable variables

A mutable variable is one whose value may change after initial assignment

side effects

Side effects are outcomes of the execution of a piece of code (for example a function) that effect the computational state of variables not explicitly named as return values for the code (function)

How can consideration of these ideas lead to more maintainable code?

Pure functions are much easier to test and more reliably composed into sophisticated functionality. In general, limiting mutation of state to as few places as possible, and limiting dependencies of code to as few (and well defined) inputs as possible, leads to more robust (bug free) code.

Question 6. In Haskell what are the names and type definitions of the MINIMAL functions that must be specified for instances of each of the following type classes:

Functor, Applicative, Foldable, Traversable and Monad?

```
class Functor (f :: * -> *) where
  (<$>) :: Functor f => (a -> b) -> f a -> f b  -- or fmap
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
class Foldable (t :: * -> *) where
```

```
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
class (Functor t, Foldable t) => Traversable (t :: * -> *) where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
```

Question 7. What is the sequence of mathematical operations that are performed because of each of the following, and what is the final result of each?

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

i) `foldl (*) 1000 [10, 5, 1]`

`((1000*10)*5)*1 = 50000`

ii) `foldr (*) 1000 [10, 5, 1]`

`10*(5*(1*1000)) = 50000`

iii) `foldl (/) 1000 [10, 5, 1]`

`((1000/10)/5)/1 = 20.0`

iv) `foldr (/) 1000 [10, 5, 1]`

`10/(5/(1/1000)) = 2.0e-3`

Question 8. What type is required for the function that needs to be placed at `_hole` in the following expression? Also give a definition for the expression using <$> and <*>.

```
sequence :: Applicative f => [f a] -> f [a]
sequence = foldr (_hole (:)) (pure [])
Applicative f => (a -> b -> c) -> f a -> f b -> f c
\f a b -> f <$> a <*> b
```

Question 9. Give examples and definitions for the following terms relating to lambda calculus:

Alpha equivalence equivalent subject to a renaming of variables

Beta normal form no further simplification by beta reduction is possible

Beta reduction application of a lambda expression to its argument

Eta conversion simplification of expressions of the form \x. f x to simply f

Combinator a function that is purely an expression of its arguments… does not contain free variables

Question 10. In MiniZinc, use a forall expression to constrain the rows of an n by n matrix to sum to the same value. Also declare that the numbers in each row are all different – but to make it more interesting don't use the built-in alldifferent constraint.

```
% sum to the same value:
var int: rowsum;
constraint
  forall (i in PuzzleRange) (
       rowsum = sum (j in PuzzleRange)(puzzle[i,j])
  );

% all different
constraint
   forall (i in PuzzleRange,j in 1..(N-1),k in (j+1)..N) (
         puzzle[i,j] != puzzle[i,k]
   );
```

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

Once again, the following short answer questions sometimes have several parts in order to cover topics in the exam without giving the actual question directly. Questions worth 4 marks on the real exam have only one part.

Question 11. (4 marks) The Math.pow function in javascript takes two arguments and raises the first to the power of the second, e.g.:

```
Math.pow(3,2)
> 9
```

Write a curried function that uses Math.pow, first with the function keyword, and then give the function again using arrow syntax. Include typescript type annotations. Show how you would use it to square the values in an array.

```
// long-form:
function pow(x:number) {
    return function(y:number) {
        return Math.pow(x,y);
    }
}
// arrow syntax:
const pow = (x:number)=> (y:number)=> Math.pow(x,y);
const flip = f => x=> y=> f(y)(x); // or just call pow from a lambda
console.log([1,2,3,4].map(flip(pow)(2)))
```

Question 12. (6 marks) Write a function that can take a function like the one you just wrote for Math.pow, and give back a binary version. Include type annotations.

```
function uncurry<T,U,V>(f: (x:T)=>(y:U)=>V) {
    return function(x,y) {
        return f(x)(y);
    }
}
```

Question 13. (8 marks total) This question is about **tail recursion** and 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 to compute the sum of a list of values with type:

```
sum :: [Integer] -> Integer
```

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

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

```
sum :: [Integer] -> Integer
sum = sum' 0
  where sum' s [] = s
        sum' s (x:xs) = sum' (s+x) xs
```

(iii) **(3 marks)** What is the benefit of tail recursion?
It allows the compiler to replace a tail recursive function with a loop that does not use stack storage.

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

TRUE = λxy. x
FALSE = λxy. y

IF = λbtf. b t f

AND = λxy. IF x y FALSE
OR = λxy. IF x TRUE y
NOT = λx. IF x FALSE TRUE

Show how the Lambda Calculus can be used to evaluate:

OR FALSE TRUE

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

Question 15. (4 marks) Consider the following JavaScript definitions:

```
const cons = head => rest => f => f(head)(rest),
      head = list => list(head=> rest => head),
      rest = list => list(head=> rest => rest);

const aList = cons(1)(cons(2)(cons(3)(null)))

const forEach = f => l => {
    if (l) {
        f(head(l));
        forEach(f)(rest(l))
    }
}
```

implement the filter function such that the following code produces the output as indicated:

```
forEach(console.log)(filter(a=>a%2==1)(aList))
>1
>3
```

```
const
    filter = f => l => l ? (
        f(head(l)) ? cons(head(l))(filter(f)(rest(l)))
                   : filter(f)(rest(l))
    ): null;
```

Question 16. (6 marks total) Consider the following TypeScript definitions:

```
interface LazySequence<T> {
  value: T;
  next(): LazySequence<T>;
}

function makeSequence<T>(getNext: (_:T)=>T, initial:T) {
  return function _next(v:T): LazySequence<T> {
    return {
      value: v,
      next: ()=>_next(getNext(v))
    }
  }(initial)
}
```

(i) What parameters would you give to makeSequence to create function which generates the following sequence:

1, -1, 2, -2, 3, -3…

(3 marks).

```
makeSequence(v=> v < 0 ? v * (-1) + 1 : -v, 1)
```

(ii) What parameters would you give to makeSequence to create function which generates the following sequence of arrays:

[1,2,3], [2,3,4], [3,4,5],…

```
makeSequence(a => a.map(v=>v+1),[1,2,3])
```

(3 marks)

Question 17. (6 marks) Write the following Haskell functions in point-free form. On separate lines, show (and name) all of the reduction and conversion steps required to achieve the point-free form *(the actual exam has only one, but I thought I would give you extra practice)*.

```
     fun1 p l = part (<p) l
fun1 p = part (<p) -- eta reduction
fun1 p = part ((>) p) -- operator fractioning
fun1 p = (part . (>)) p -- compose
fun1 = part . (>) -- eta reduction

 fun2 a b c = (a * b) + c
```

```
fun2 a b c = (+) (a*b) c
fun2 a b = (+) (a*b) -- eta reduction
fun2 a b = (+) ((*) a b) -- operator fractioning
fun2 a b = (+) (((*) a) b) -- making precedence explicit
fun2 a b = ((+) . (((*) a)) b -- compose
fun2 a = (+) . ((*) a) -- eta reduction
fun2 a = ((+) .) . (*) a -- compose
fun2 = ((+) . ) . (*) -- eta reduction
```

```
     fun3 a b = a `div` (g b)
fun3 a b = div a (g b)
fun3 a b = ((div a) . g) b -- compose
fun3 a = (div a) . g -- eta reduction
fun3 a = (.g) (div a) -- operator fractioning on (.)
fun3 = (.g) . div -- eta reduction
```

Question 18. (6 marks) Sudoku puzzles typically involve completing a 9 by 9 grid of digits (1..9) such that in each row, column and 3 by 3 subsquare each digit appears only once. For example:

	6	8	4		1		7	
				8	5		3	
	2	6	8		9		4	
		7				9		
	5		1		6	3	2	
	4		6	1				
	3		2		7	6	9	

In the MiniZinc code below which models and solves the above puzzle, complete the constraints for columns and sub-squares.

```
start=[|
  0, 0, 0, 0, 0, 0, 0, 0, 0|
  0, 6, 8, 4, 0, 1, 0, 7, 0|
  0, 0, 0, 0, 8, 5, 0, 3, 0|
  0, 2, 6, 8, 0, 9, 0, 4, 0|
  0, 0, 7, 0, 0, 0, 9, 0, 0|
  0, 5, 0, 1, 0, 6, 3, 2, 0|
  0, 4, 0, 6, 1, 0, 0, 0, 0|
  0, 3, 0, 2, 0, 7, 6, 9, 0|
  0, 0, 0, 0, 0, 0, 0, 0, 0|]

int: S = 3;
int: N = S * S;

set of int: PuzzleRange = 1..N;
set of int: SubSquareRange = 1..S;

array[1..N,1..N] of 0..N: start; %% initial board 0 = empty
```

```
array[1..N,1..N] of var PuzzleRange: puzzle;

% fill initial board
constraint forall(i,j in PuzzleRange)(
  if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );

% All different in rows
constraint forall (i in PuzzleRange) (
  alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% All different in columns.
constraint forall (j in PuzzleRange) (
  alldifferent( [ puzzle[i,j] | i in PuzzleRange ]));

% All different in sub-squares:
constraint
     forall (a, o in SubSquareRange)(
          alldifferent( [ puzzle[(a-1) *S + a1, (o-1)*S + o1] |
               a1, o1 in SubSquareRange ] ) );
solve satisfy;
```

Question 19. (6 marks) How can Haskell ADTs be used as the product of a parser? How do they relate to the grammar of a language? How does such an ADT relate to an ADT? Refer to the examples discussed in lectures and tutes in your answer to the above.

The ADT is typically defined with constructors that correspond to the terminals and non-terminals in the BNF grammar.
A parse tree is has internal nodes (roughly) corresponding to the non-terminals of the grammar.
Examples that we looked at of parsers that generated parse trees were JSON and calculator.

Question 20. (8 marks total)

(i) **(4 marks)** A Rose-tree can store a value, and a list of sub-trees in its nodes – or it can be Nil. Give a generic algebraic data type for such a rose tree that also includes terms in its internal nodes for the minimum and maximum values in any of its subtrees.

(ii) **(4 marks)** Consider a binary search tree with ordered keys and a value associated with each key. Describe in words and diagrams how a pure function can change the value (not the key) stored at one of the leaves in better than linear time and memory.

(i) `data RoseTreeMinMax a :: Eq a, Ord a => Nil | Node a a a [RoseTreeMinMax a]`

(ii) We are using the above algebraic data type to store a binary tree, so we assume the list of children at each node has length at most two.

The concrete type for a will be a key-value pair, that instances the Ord and Eq TypeClasses with compare and (==) functions that tests the key (first member of the pair).
The nodes are stored such that the left child of any node (first element in child list) has key smaller than the parent. The right child (second element in child list) has key greater than or equal to the parent's key.
Assume the tree is balanced and therefore that the height of the tree is O(log n). Thus, since the nodes are also ordered, we can find a node with any key in O(log n) time.
A pure function cannot modify any of the nodes in place.
Thus, to modify a value we have to create a new node, and also create new parent nodes all the way to the root. However, there are at most O(log n) of these, such that the time and space required for the update is still O(log n).
This answer does not give any guarantees about the time complexity of subsequent update operations. If we did we would also need an efficient operation to restore the tree's order.

END OF EXAMINATION