| PAPER CODE NO. | EXAMINER: John Fearnley | Tel. No. 54265 |
|---|---|---|
| **COMP105** | DEPARTMENT: Computer Science | |

UNIVERSITY OF
LIVERPOOL

# PRACTICE EXAMINATION
# Programming Language Paradigms

**TIME ALLOWED : Three Hours**

---

**INSTRUCTIONS TO CANDIDATES**

This is a computer-based semi-open book exam. You may access all materials on the COMP105 Canvas course, and you may access Codegrade. You may not access any material elsewhere on the Internet. You may not use pre-written notes or code, and you may not use your own hardware.

Make sure that you are logged in to Canvas at the start of the exam. You may use a mobile device in order to do this. After this, the use of mobile devices during the exam is prohibited. If you become logged out of Canvas during the exam, then notify an invigilator, and they will supervise your use of a mobile device while you log back in.

Fill out your answers in a file named `Practice.hs`. Your answers must be submitted to Codegrade in the "Practice Exam" assessment that can be found in the COMP105 Canvas course.

The exam consists of five sections. Each section is weighted equally, and each question within each section is weighted equally. Each question has one visible test, which will allow you to confirm that your function has the correct type. The visible test doest not award any marks. All marks will be derived from the hidden tests, which will not be revealed to you during the exam.

You may resubmit to Codegrade as many times as you like during the exam. The last submission that you make is the one that will be marked. If you do not submit a file to Codegrade during the exam then you will be marked as absent and receive a mark of zero.

You may not import any modules into `Exam.hs` except `System.Environment`. You may write and use helper functions in your solutions. Some questions have particular requirements such as "you must use recursion" or "you may not use library functions". If your solution does not meet these requirements then you will receive a mark of zero for that question.

A requirement to "not use library functions" means that you may not call a library function that is prefix by default. You may, in any question, use library functions that are infix by default (eg. `+`, `*`, `:`, `++`), Haskell syntax (eg. `if` expressions, `let` expressions, `where clauses`), pattern matching, and list ranges.

UNIVERSITY OF LIVERPOOL

## Section A – Haskell Basics

1. Write a function `expression :: Float -> Float -> Float -> Float` that takes floats `x`, `y`, and `z` and returns $x + y$ divided by $z$. For example:

```
ghci> expression 2 4 3
2.0
```

2. Write a function `non_negative :: Int -> Int` that takes an integer `x`. If `x` is greater than or equal to zero, then the function should return `x`, otherwise it should return zero. For example:

```
ghci> non_negative 3
3
ghci> non_negative (-3)
0
```

3. Write a function `get_middles :: (Int, Int, Int) -> (Int, Int, Int) -> (Int, Int)` that takes two three-element tuples, and returns a two-element tuple that contains the middle element from each of the two inputs. For example:

```
ghci> get_middles (1, 2, 3) (4, 5, 6)
(2,5)
```

4. Write a function `fourth_or_zero :: [Int] -> Int` that takes a list of integers and outputs the fourth element of the list. If the list has fewer than four elements, then the function should output zero instead. For example:

```
ghci> fourth_or_zero [1,2,3,4]
4
ghci> fourth_or_zero [1,2,3]
0
```

5. Use a list comprehension to write a function `remove_aes :: [Char] -> [Char]` that takes a string, and returns a copy of that string with all instances of the character `'a'` removed, and all instances of the character `'e'` removed. For example:

```
ghci> remove_aes "character"
"chrctr"
```

   Your solution must use a list comprehension. You may not use recursion or library functions.

UNIVERSITY OF LIVERPOOL

## Section B – Recursion

You must use recursion to solve **all** questions in Section B, and you may not use library functions except `head` and `tail` unless otherwise permitted. You may not use list comprehensions in your solutions for this section.

1. Write a function `sum_squares :: Int -> Int` that takes an integer `x` and returns the sum of the squares between 0 and `x` inclusive. For example

   ```
   ghci> sum_squares 3
   14
   ```

   Here we summed $0^2 + 1^2 + 2^2 + 3^2 = 14$.

2. Write a function `square_evens :: [Int] -> [Int]` that takes a list of integers. Every even number in the input list should be squared, while every odd number should be removed. You may use the `mod` function in your solution. For example:

   ```
   ghci> square_evens [1,2,3,4]
   [4,16]
   ```

3. Write a function `count_xys :: [Char] -> (Int, Int)` that takes a string, and returns a two-element tuple that contains the number of times that `'x'` appears in the string followed by the number of times that `'y'` appears in the string. For example:

   ```
   ghci> count_xys "xyzzy"
   (1,2)
   ```

4. A *peak* in a sequence of numbers is a consecutive sequence $x$, $y$, $z$ where $x < y$ and $y > z$. Write a function `find_peaks :: [Int] -> [[Int]]` that takes a list of integers, and outputs a list containing all of the peaks in the input list in the order that they appear. For example:

   ```
   ghci> find_peaks [1,2,3,2,1,2,3,4,3]
   [[2,3,2],[3,4,3]]
   ```

5. Write a function `extract_brackets :: [Char] -> [[Char]]` that takes a string and outputs a list of all substrings that lie between a pair of brackets. You may use `takeWhile`, `dropWhile`, and `length` in your solution. For example:

   ```
   ghci> extract_brackets "ab(cde)fg(hijk)"
   ["cde","hijk"]
   ```

   You can assume that brackets always come in pairs and are never nested, meaning that there is always a `')'` character between any two `'('` characters.

UNIVERSITY OF LIVERPOOL

## Section C – Higher Order Functions

You may **not** use recursion or list comprehensions in your solutions for Section C. You may use library functions.

1. Write a function `get_first_two :: [[Char]] -> [(Char, Char)]` that takes a list of strings, and returns a list of tuples, where each tuple contains the first two characters of the corresponding string. Your solution must use `map`. For example:

   ```
   ghci> get_first_two ["abcd", "efg", "hi"]
   [('a','b'),('e','f'),('h','i')]
   ```

   You can assume that each string in the input list has at least two characters.

2. Write a function `both_true :: [(Bool, Bool)] -> [(Bool, Bool)]` that takes a list of two-element tuples, and returns a list that contains only the tuples in which both elements are `True`. Your solution must use `filter`. For example:

   ```
   ghci> both_true [(True, True), (True, False), (False, False)]
   [(True,True)]
   ```

3. Write a function `split_evens :: [Int] -> ([Int], [Int])` that takes a list of integers, and returns a two-element tuple, where the first element of the tuple contains all of the even numbers in the input list, and the second element contains all of the odd numbers. Your solution must use `foldr`. For example:

   ```
   ghci> split_evens [1,2,3,4,5]
   ([2,4],[1,3,5])
   ```

4. Write a function `get_indices :: [[Char]] -> [Int] -> [Char]` that takes a list of strings `xs`, and a list of integers `ys`. The elements of `ys` should be thought of as indexes for the corresponding strings in `xs`. The output list should contain the requested elements of `xs`, or the character `'!'` if the index is too large. Your solution must use `zip` or `zipWith`. For example:

   ```
   ghci> get_indices ["abc", "def", "hij"] [0, 1, 3]
   "ae!"
   ```

   Here we asked for the first element of `"abc"`, the second element of `"def"`, and the fourth element of `"hij"`.

5. Write a function `total_sales :: [Char] -> [Char]` that takes a string as an argument. The string contains lists of transaction values for a number of days, where each day appears on a separate line. The function should output a report giving the total amount transacted on each day. For example:

   ```
   ghci> total_sales "Monday 10 20 30\nTuesday 1 2 3\n"
   "Monday: 60\nTuesday: 6\n"
   ```

   You can assume that the input contains a number of lines that each end with a `'\n'` character. You can also assume that each line starts with the name of a day, which is then followed by a list of integers each separated by exactly one space character.

UNIVERSITY OF LIVERPOOL

## Section D – Custom Types and IO

1. Create a custom type `DoubleMaybe` that is parameterized by type variables `a` and `b`. The type should have three constructors: the constructor `Two` should contain a value of type `a` and a value of type `b` in that order, the constructor `One` should contain a value of type `a`, and the constructor `DoubleNothing` should contain no values. Make the `DoubleMaybe` type an instance of `Show`, `Read`, and `Eq`.

2. Write a function `sum_double_maybe :: Num a => DoubleMaybe a a -> Maybe a` that takes a `DoubleMaybe` value that contains numbers, and sums all of the numbers in the given constructor. If `DoubleNothing` is passed to the function, it should return `Nothing`, otherwise it should return the sum in a `Just` constructor. For example:

```
ghci> sum_double_maybe (Two 1 2)
Just 3
ghci> sum_double_maybe (One 3)
Just 3
ghci> sum_double_maybe DoubleNothing
Nothing
```

3. Using the following type definition

```
data DTree a = Leaf a | Branch a (DTree a) (DTree a) deriving (Show, Read)
```

   write a function `tree_elem :: Eq a => a -> DTree a -> Bool` that takes a value `x` and a tree, and returns `True` if the tree contains `x`, and `False` otherwise. For example:

```
ghci> tree_elem 1 (Branch 1 (Branch 2 (Leaf 3) (Leaf 4)) (Leaf 5))
True
```

4. Write an IO action `print_string_or_bool :: [Char] -> Bool -> IO ()` that takes a string and a boolean as arguments. It should ask the user to input a string. If the user inputs `string`, then the string should be printed, otherwise the boolean should be printed. For example:

```
ghci> print_string_or_bool "hi" True
string
hi
ghci> print_string_or_bool "hi" True
something else
True
```

   Here the second and fifth lines were entered by the user.

**5.** Write an IO action `sum_input :: IO Integer` that repeatedly asks the user to input integers until the user inputs `0`. The IO action should return the sum of the integers entered by the user. For example:

```
ghci> sum_input
1
2
3
0
6
```

Here the second, third, fourth, and fifth lines were entered by the user. You can assume that the user always enters a valid integer.

## Section E – Challenge Question

Write a program (by implementing `main` in `Practice.hs`) that implements the following specification. The program will compute the total cost of constructing an item, given prices for the items and parts that make up that item.

The program will take a single command line argument specifying the path to a configuration file. An example configuration file is as follows.

```
REQUIRES car wheel wheel wheel wheel engine
REQUIRES wheel tire hub
REQUIRES engine cylinder cylinder piston piston
PART tire 20
PART hub 40
PART cylinder 100
PART piston 120
```

Each line of the configuration file will either be a part specification or a requires specification. A part specification starts with `PART` followed by the name of the part and an integer giving the cost of that part. A requires specification starts with `REQUIRES` followed by the name of the item, followed by a list of items that are needed to make that item. So in the example above, the first line specifies that a car is made up of four wheels and an engine.

The program will ask the user to input a string specifying an item. The program should then compute the total cost required to construct that item and print that cost to the terminal.

If the configuration given above is saved as a file named `config`, then an example interaction with the program is:

```
> Practice config
wheel
60
```

Here the first line shows the program being invoked by the user at the operating system command line, and the second line was entered by the user.

You may assume that the file path passed to the program is always a valid file, and that the file always meets the specification given above. In particular, you may assume that each line of the file consists of a sequence of words that are each separated by exactly one space, and that each line is ended by a `'\n'` character. You may also assume that the user always enters a valid item. Finally, you may assume that the configuration file does not contain any cycles, meaning that it is not possible for an item to require itself as part of its construction.