Programming in Haskell 2nd edition Hutton Test Bank

SKU: 9781316626221-TEST-BANK

The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 1 MODULE, SPRING SEMESTER 2011-2012

FUNCTIONAL PROGRAMMING

Time allowed TWO hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced.

Answer QUESTION ONE and THREE other questions

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject specific translation dictionaries are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

DO NOT turn examination paper over until instructed to do so

ADDITIONAL MATERIAL: Haskell Standard Prelude

Question 1 (Compulsory)

Select ONE answer for each section.

There is no negative marking for incorrect answers.

a) The expression `["False","True"]` has type:

i) `[a]`
ii) `[Bool]`
iii) `[String]`
iv) `[Bool,Bool]`
v) `[String,String]` (3 marks)

b) Which of the following is an invalid list in Haskell:

i) `[[1,2,3,4]]`
ii) `[1,[2,3],4]`
iii) `[[1,2],[3,4]]`
iv) `[[1],[2,3],[4]]`
v) `[[1],[2],[3],[4]]` (3 marks)

c) Evaluating `[(x,y) | x <- [1,2], y <- [1,2]]` gives:

i) `([1,2],[1,2])`
ii) `[(1,2),(1,2)]`
iii) `[(1,1),(2,2)]`
iv) `[(1,1),(1,2),(2,1),(2,2)]`
v) `[(1,1),(2,1),(1,2),(2,2)]` (3 marks)

d) A function of type `(Int -> Int) -> Int`:

i) Takes a function as its argument
ii) Takes two arguments one at a time
iii) Takes a pair of arguments
iv) Returns a function as its result
v) Returns a pair of results (3 marks)

e) Evaluating `sum [x | x <- [1..10], even x]` gives:

i) An error
ii) `10`
iii) `25`
iv) `30`
v) `55` (3 marks)

f) Evaluating `zip [1,2] ['a','b','c']` gives:

i) An error
ii) `([1,2],['a','b','c'])`
iii) `[(1,'a'),(2,'b')]`
iv) `[(1,'a'),(2,'b'),(2,'c')]`
v) `[(1,'a'),(2,'b'),(3,'c')]` (3 marks)

g) Which of the following statements about Haskell is true:

i) Function application brackets to the left
ii) All programs are guaranteed to terminate
iii) All lists must be of finite length
iv) Recursive functions must have a base case
v) Type errors can occur at run-time (3 marks)

h) The expression `Node (Leaf 1) (Leaf 2)` is a value of the datatype:

i) `data Tree = Node | Leaf Int`
ii) `data Tree = Leaf Int | Node Int Int`
iii) `data Tree = Leaf Tree | Node Int Int`
iv) `data Tree = Leaf Int | Node Tree Tree`
v) `data Tree = Leaf Tree | Node Tree Tree` (4 marks)

Question 2:

a) What are the types of the following expressions? (5 marks)

```
"Haskell"
[False,True,False]
(False,'a')
(['a','b'],[False,True])
filter
```

b) Define the following library functions using recursion: (8 marks)

```
product :: [Int] -> Int
length  :: [a] -> Int
reverse :: [a] -> [a]
map     :: (a -> b) -> [a] -> [b]
```

c) Using the definition

```
fac   :: Int -> Int
fac 0  = 1
fac n  = n * fac (n-1)
```

show how the expression `fac 3` is evaluated. (4 marks)

d) Given the alternative definition

```
fac       :: Int -> Int
fac n      = accum 1 n

accum     :: Int -> Int -> Int
accum x 0  = x
accum x y  = accum (x*y) (y-1)
```

in terms of an auxiliary function `accum` that uses an extra argument to accumulate the result, show how `fac 3` is now evaluated. (4 marks)

e) Explain using your answers to the two previous parts why the original definition for `fac` is inefficient in terms of memory usage, and how the alternative definition resolves this problem. (4 marks)

Question 3:

a) Give concise English definitions for the following terms: (5 marks)

Recursive function

Curried function

Higher-order function

Polymorphic function

Overloaded function

b) Give one example function definition from the Haskell Standard Prelude for each of the five terms that are listed above. You may not use the same example function for more than one answer. (5 marks)

c) The sum of the integers between 1 and 100 can be expressed using the list comprehension notation as `sum [x | x <- [1..100]]`. Express each of the following using a list comprehension: (8 marks)

Sum of the squares of the integers between 1 and 100

Product of the even integers between 1 and 100

List of all pairs of integers between 1 and 100

Sum of the products of all pairs of integers between 1 and 100

d) Suppose you are given a function `divides :: Int -> Int -> Bool` that decides if one integer is divisible by another, e.g. `divides 15 2` is `False` and `divides 15 3` is `True`. Using a list comprehension, define a function `divisors :: Int -> [Int]` that returns the divisors of a natural number. For example, `divisors 15` should give `[1,3,5,15]`. (4 marks)

e) A natural number greater than one is *prime* if its only divisors are one and itself. Using `divisors`, define a function `prime :: Int -> Bool` that decides if a natural number is prime or not. (3 marks)

Question 4:

Suppose that you have been invited to write an article for a professional computing magazine on the *benefits that Haskell brings to programmers.* Write a short article on this topic, illustrating each of the benefits that you mention with a small example written in Haskell. (25 marks)

Question 5:

a) Complete the missing parts `???` in the following definition for a recursive function that inserts an integer into the correct position in a sorted list. For example, `insert 3 [1,2,4,5]` should give `[1,2,3,4,5]`. (6 marks)

```
insert          :: Int -> [Int] -> [Int]
insert x []      = ???
insert x (y:ys) = if x <= y then
                     ???
                  else
                     ???
```

b) Show how your definition evaluates `insert 3 [1,2,4,5]`. (4 marks)

c) Using `insert`, complete the following definition for a recursive function that sorts a list of integers using *insertion sort*, in which the empty list is already sorted, and any non-empty list is sorted by inserting the head into the list that results from sorting its tail. (4 marks)

```
isort         :: [Int] -> [Int]
isort []       = ???
isort (x:xs)   = ???
```

d) Show how your definition evaluates `isort [3,2,1]`. There is no need to show how each application of `insert` is evaluated. (5 marks)

e) Complete the missing parts in the following definition for a recursive function that implements *quicksort*, in which the empty list is already sorted, and any non-empty list is sorted by sorting the tail values `<=` the head, sorting the tail values `>` the head, and appending the resulting sorted lists on either side of the head value. (6 marks)

```
qsort         :: [Int] -> [Int]
qsort []       = ???
qsort (x:xs)   = qsort ??? ++ ??? ++ qsort ???
```

The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, SPRING SEMESTER 2011-2012

ADVANCED FUNCTIONAL PROGRAMMING

Time allowed TWO hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced.

Answer FOUR out of five questions

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject specific translation dictionaries are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

DO NOT turn examination paper over until instructed to do so

ADDITIONAL MATERIAL: Haskell Standard Prelude

Question 1:

Write a short introduction to the *use of monads to structure Haskell programs*, using the example of writing a function

$$eval \quad :: \quad Expr \rightarrow Maybe\ Int$$

that evaluates expressions in the following simple language, in which attempting to divide by zero results in the value *Nothing*:

$$\textbf{data}\ Expr \quad = \quad Val\ Int \mid Div\ Expr\ Expr$$

You may assume that your audience is familiar with the basics of Haskell, but has no previous experience with monads. (25)

Question 2:

a) Show how the notion of a *state transformer* can be represented in Haskell as a parameterised type ST, and explain your definition. (4)

b) Define appropriate functions *return* and $\gg\!=$ that make ST into a monad, and explain your definitions with the aid of pictures. (6)

c) Given the type definition

$$\textbf{data}\ Tree\ a \quad = \quad Leaf\ a \mid Node\ (Tree\ a)\ (Tree\ a)$$

define a *non-monadic* function

$$label \quad :: \quad Tree\ a \rightarrow Int \rightarrow (Tree\ Int, Int)$$

that replaces every leaf value in such a tree with a unique or *fresh* integer, by taking a next fresh integer as an additional argument, and returning the next fresh integer as an additional result. (4)

d) Show how the *label* function can be redefined in a monadic manner by exploiting the fact that ST forms a monad. (8)

e) Why is the monadic definition for *label* preferable? (3)

Question 3:

a) Using equational reasoning, and being explicit about any arithmetic properties being exploited in each step, prove that: (4)

$$(x + a)(x + b) \quad = \quad xx + (a + b)x + ab$$

b) Given the definitions

$$\begin{array}{lcl} (+\!\!+) & :: & [a] \to [a] \to [a] \\ [\,] +\!\!+ ys & = & ys \\ (x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys) \end{array}$$

$$\begin{array}{lcl} reverse & :: & [a] \to [a] \\ reverse\ [\,] & = & [\,] \\ reverse\ (x : xs) & = & reverse\ xs +\!\!+ [x] \end{array}$$

and using the fact that $+\!\!+$ is associative, prove that: (6)

$$reverse\ (xs +\!\!+ ys) \quad = \quad reverse\ ys +\!\!+ reverse\ xs$$

c) Explain why the above definition for *reverse* is inefficient, and show how a more efficient version can be defined using the technique of *accumulation*, illustrating your new definition using a simple example. (6)

d) Given the definitions

$$\begin{array}{lcl} sum & :: & [Int] \to Int \\ sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \end{array}$$

$$\begin{array}{lcl} length & = & [a] \to Int \\ length\ [\,] & = & 0 \\ length\ (x : xs) & = & 1 + length\ xs \end{array}$$

define a function

$$sumlen \quad :: \quad [Int] \to (Int, Int)$$

that satisfies the specification $sumlen\ xs = (sum\ xs, length\ xs)$, but only makes a single traversal over the list, rather than two. (4)

e) Prove that your definition for *sumlen* satisfies its specification. (5)

Question 4:

a) Given the function definition

$$\begin{array}{lcl} sum & :: & [Int] \to Int \\ sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \end{array}$$

explain with the aid of a simple example why this definition is potentially inefficient in terms of memory usage. (3)

b) Given the specification $sum'\ xs\ n = n + sum\ xs$, calculate a recursive definition for sum' using *constructive induction* on xs. You may assume standard arithmetic properties of addition. (5)

c) Given the revised definition $sum\ xs = sum'\ xs\ 0$, explain using your previous example why this definition is potentially more efficient. (3)

d) Given the type declarations

$$\begin{array}{lcl} \textbf{data}\ Expr & = & Val\ Int \mid Add\ Expr\ Expr \\ \textbf{type}\ Stack & = & [Int] \\ \textbf{type}\ Code & = & [Op] \\ \textbf{data}\ Op & = & PUSH\ Int \mid ADD \end{array}$$

define three functions

$$eval :: Expr \to Int$$

$$comp :: Expr \to Code$$

$$exec :: Code \to Stack \to Stack$$

that evaluate an expression to an integer value, compile an expression to code, and execute code using an initial stack to give a final stack. (6)

e) Assuming the distributivity lemma

$$exec\ (xs +\!\!+ ys)\ s = exec\ ys\ (exec\ xs\ s)$$

verify the compiler correctness property below by induction on e, justifying each step in your equational reasoning with a short hint. (8)

$$exec\ (comp\ e)\ s = (eval\ e) : s$$

Question 5:

Consider the following representation of Sudoku grids:

$$\begin{array}{lcl} \textbf{type } \mathit{Grid} & = & \mathit{Matrix}\ \mathit{Int} \\ \textbf{type } \mathit{Matrix}\ a & = & [\mathit{Row}\ a] \\ \textbf{type } \mathit{Row}\ a & = & [a] \end{array}$$

a) Suppose that you are given functions

$$\begin{array}{lcl} \mathit{rows} & :: & \mathit{Matrix}\ a \rightarrow [\mathit{Row}\ a] \\ \mathit{cols} & :: & \mathit{Matrix}\ a \rightarrow [\mathit{Row}\ a] \\ \mathit{boxs} & :: & \mathit{Matrix}\ a \rightarrow [\mathit{Row}\ a] \\ \mathit{complete} & :: & \mathit{Row}\ a \rightarrow \mathit{Bool} \end{array}$$

that extract the rows, columns and boxes from a matrix, and decide if a row is complete (contains each of the numbers 1 to 9 exactly once). Using these functions, define a function $\mathit{valid} :: \mathit{Grid} \rightarrow \mathit{Bool}$ that decides if all the rows, columns and boxes in a grid are complete. (4)

b) Define a function $\mathit{choices} :: \mathit{Grid} \rightarrow \mathit{Matrix}\ [\mathit{Int}]$ that replaces each zero value (representing a blank entry) in the grid by the list of choices [1..9] for that value, and each non-zero value n by the singleton list $[n]$. (4)

c) Define a function $\mathit{cp} :: [[a]] \rightarrow [[a]]$ that returns the cartesian product of a list of lists, e.g. $\mathit{cp}\ [[1,2],[3,4]] = [[1,3],[1,4],[2,3],[2,4]]$. (4)

d) Using *cp*, define a function $\mathit{collapse} :: \mathit{Matrix}\ [a] \rightarrow [\mathit{Matrix}\ a]$ that collapses a matrix of choices into a choice of matrices. (3)

e) Using your answers to the previous parts of this question, define a function $\mathit{solve} :: \mathit{Grid} \rightarrow [\mathit{Grid}]$ that solves Sudoku puzzles. Explain why this function is too inefficient to be practically useful. (5)

f) Suppose that you are given a function

$$\mathit{prune} :: \mathit{Matrix}\ [\mathit{Int}] \rightarrow \mathit{Matrix}\ [\mathit{Int}]$$

that removes all choices that already occur as single entries in the associated row, column or box of a matrix. Using this function, define a new version of *solve* that can instantly solve any "easy" Sudoku puzzle that only requires the repeated application of the basic constraints of the game. (5)