

SAMPLE EXAM

HASKELL

1. Assume the variable **x** is bound to a list of numbers. Enter a single Haskell expression confined to a single line that will return a list of the squares of the elements of **x**. For example, if **x** is **[1,2,3,2,1]**, your expression should return **[1,4,9,4,1]**. You may use any Haskell Prelude functions and types, but you cannot use any other libraries and you cannot define any new functions. You should assume that the variable **x** is already defined.
2. Assume the variable **x** is bound to a list of numbers (**Ints**, **Doubles**, etc). Enter a single Haskell expression confined to a single line that will return the list **x** with all the zeros removed from the beginning and end of the list, but other zeros retained. For example, if **x** is **[0,0,0,2,0,7,0,0]**, your expression should return **[2,0,7]**. You may use any Haskell Prelude functions and types, but you cannot use any other libraries and you cannot define any new functions. You should assume that the variable **x** is already defined.
3. Assume the variable **x** is bound to a number (an **Int**, **Double**, etc), and **c** is bound to a list of coefficients. Enter a single Haskell expression confined to a single line that will return the value of the polynomial defined by **x** and **c**. That is, it should result in the first element of **c** plus the second element times **x** plus the third element times **x** squared plus the fourth times **x** cubed, etc. For example, if **x** is **2** and **c** is **[1,3,5]**, your expression should return $1 + 3 * x + 5 * x * x = 27$. You may use any Haskell Prelude functions and types, but you cannot use any other libraries and you cannot define any new functions. You should assume that the variables **x** and **c** are already defined.
4. Design a type **ChessPiece** to represent a piece in the game of chess. Each chess piece is either black or white, and is either a king, queen, rook, bishop, knight, or pawn. Ensure that, if **p** is a **ChessPiece**, then **show p** returns a *two-character* string, where the first character is either **B** for black or **W** for white, and the second is one of: **K** for king, **Q** for queen, **R** for rook, **B** for bishop, **N** for knight, or **P** for pawn.

Also define a function

`toChessPiece :: String -> Maybe ChessPiece`

that takes a string as input and returns **Just** a chess piece if the string consists of a two character string matching the above specification, ignoring any space characters in the input string, and **Nothing** otherwise. So for example, `toChessPiece " WN "` should return **Just** the representation of a white knight, while `toChessPiece " NW "` should return **Nothing**.

Feel free to define as many helper types and functions as you like to assist your implementation.

You will be assessed both on the correctness of your implementation and the quality of your type definition. You should make your type robust, so that any program manipulating chess pieces will have compile-time (*not* run-time) errors if any invalid chess pieces or aspects of them are used in the program.

```
module ChessPiece (ChessPiece, toChessPiece) where
-- Define your ChessPiece type here. Do not change the above module line.
```

5. Write a Haskell function

```
divideList :: (a -> Bool) -> [a] -> ([a],[a])
```

such that **divideList p lst** splits the list **lst** into two parts: the initial elements of the list all of which satisfy **p**, and then the remainder of the list. Thus if **(front,back) = divideList p lst**, then **lst == front ++ back**, and every element of **front** satisfies **p**, and if **back** is non-empty, then **p (head back)** is **False**. Note that **front** or **back** (or both) may be empty.

For example: **divideList (<5) [2,4,6,1,3,9]** should yield **([2,4],[6,1,3,9])**.

6. Given the Haskell type

```
data BST a = Empty | Node (BST a) a (BST a)
```

representing a binary search tree, write a Haskell function **isBalanced** that takes a **BST** as input and returns a **Bool** indicating whether or not the input is balanced.

A **BST** is considered to be balanced if no path from the root of the tree to any leaf (that is, to an **Empty** constructor) is more than one node longer than any other path from root to leaf in that tree. In other words, the difference between the **maximum** and **minimum** depth of the tree is no more than one.

For example, these trees are balanced:

- Node (Node Empty 1 Empty) 2 (Node Empty 3 Empty)
- Node (Node (Node Empty 1 Empty) 2 Empty) 3 (Node Empty 4 Empty)
- Empty

and these are not:

- Node (Node (Node Empty 1 Empty) 2 Empty) 3 Empty
- Node (Node Empty 1 (Node Empty 2 (Node Empty 3 Empty))) 4 (Node Empty 5 (Node Empty 6 (Node Empty 7 Empty)))

You may use any standard Haskell prelude functions you like in your implementation.

Logic Program Semantics

7. Consider this Prolog program:

```
p(a,b). p(b,c). p(c,d).
```

```
q(X,Y) :- p(X,Y).
```

```
q(X,Y) :- p(Y,X).
```

```
r(X,Z) :- p(X,Z).
```

```
r(X,Z) :- p(X,Y), r(Y,Z).
```

List all ground unit clauses that form the semantics for the above Prolog program, using proper Prolog syntax. Each correct clause is with an equal fraction of the available marks, and each extra (incorrect) clause is leads to a deduction of the same fraction of the available marks.

8. Consider the following Prolog predicates definining a database of companies, employees, and their relationships.

company(Name, N, Revenue) holds when **Name** is a company with **N** employees and an annual revenue of **Revenue**.

employee(E, C) holds when **E** is an employee of the company **C**.

knows(X, Y) holds when person **X** knows person **Y**.

A **sample** of this database is shown below:

```
company('Babbling Books', 500, 10000000).
```

```
company('Crafty Crafts', 5, 250000).
```

```
company('Hatties Hats', 25, 10000).
```

```
employee(mary, 'Babbling Books').
```

```
employee(julie, 'Babbling Books').
```

```
employee(michelle, 'Hatties Hats').
```

```
employee(mary, 'Hatties Hats').
```

```
employee(javier, 'Crafty Crafts').
```

```
knows(javier, michelle).
```

Write a Prolog query to find all the N such that N is the name of a company that employs between 50 and 2000 employees (2 marks).

Grok will print all possible bindings for variable **N** in the output pane (to the right) when you run your query. If you do not see any output when running your query, this means that you have either not used the variable **N** in your query or your query does not bind **N** to any values.

9. Consider the following Prolog predicates defining a database of companies, employees, and their relationships.

company(Name, N, Revenue) holds when **Name** is a company with **N** employees and an annual revenue of **Revenue**.

employee(E, C) holds when **E** is an employee of the company **C**.

knows(X, Y) holds when person **X** knows person **Y**.

A **sample** of this database is shown below:

```
company('Babbling Books', 500, 10000000).  
company('Crafty Crafts', 5, 250000).  
company('Hatties Hats', 25, 10000).
```

```
employee(mary, 'Babbling Books').  
employee(julie, 'Babbling Books').  
employee(michelle, 'Hatties Hats').  
employee(mary, 'Hatties Hats').  
employee(javier, 'Crafty Crafts').
```

```
knows(javier, michelle).
```

Write a Prolog query to find all N where N is the name of an employee who works (is employed) at two different companies (2 marks).

Grok will print all possible bindings for variable **N** in the output pane (to the right) when you run your query. If you do not see any output when running your query, this means that you have either not used the variable **N** in your query or your query does not bind **N** to any values.

10. Consider the following Prolog predicates defining a database of companies, employees, and their relationships.

company(Name, N, Revenue) holds when **Name** is a company with **N** employees and an annual revenue of **Revenue**.

employee(E, C) holds when **E** is an employee of the company **C**.

knows(X, Y) holds when person **X** knows person **Y**.

A **sample** of this database is shown below:

```
company('Babbling Books', 500, 10000000).  
company('Crafty Crafts', 5, 250000).  
company('Hatties Hats', 25, 10000).
```

```
employee(mary, 'Babbling Books').  
employee(julie, 'Babbling Books').  
employee(michelle, 'Hatties Hats').  
employee(mary, 'Hatties Hats').  
employee(javier, 'Crafty Crafts').
```

```
knows(javier, michelle).
```

Write a Prolog query to find a list L of employee pairs (in the form Name1-Name2), where each pair represents two employees that know each other but who work at different companies (2 marks).

Grok will print all possible bindings for variable **L** in the output pane (to the right) when you run your query. If you do not see any output when running your query, this means that you have either not used the variable **L** in your query or your query does not bind **L** to any values.

11. Write a Prolog predicate **words/2** such that **words(X, Y)** holds when **X** is a list of characters and **Y** is a list of words, where each word is a list of characters not containing any space characters (' '), and **X** is the result of appending together all the words on **Y**, in the order they appear in **Y**, with a single space character between words.

This predicate can be used to split a line of text into a list of words (as long as there is only one space between words), or to put a list of words together into a line of text. For example, these queries should produce these, and only these, solutions:

```
?- words([t,h,i,s,' ',i,s,' ',a,' ',t,e,s,t], X).
```

```
X = [[t,h,i,s],[i,s],[a],[t,e,s,t]]
```

```
?- words(X, [[t,h,i,s],[i,s],[a],[t,e,s,t]]).
```

```
X = [t,h,i,s,' ',i,s,' ',a,' ',t,e,s,t]
```

Your predicate should work in any mode in which at least one of the arguments is ground. It need not work with both arguments unbound. An implementation that works correctly, but only in the mode where the first argument is ground, will be eligible to receive 80% of the marks.

You may use any Prolog built-in or library predicates.

