Programming Task #3: A Role-Playing Adventure

Version 1.0: 11 April 2022

Academic Integrity

As a reminder of the course honor policy, **all work submitted for this programming task must be yours alone**. Refer to the *General Information about Programming Tasks* section of Blackboard for more details about what is permitted and not permitted.

Learning Targets

This programming task relates directly to the following Learning Target(s):

- **P.3**: I can use pattern matching and algebraic datatypes to write data-directed functions/programs.

Before You Start: Read This

Grab a copy of the file `RPGdefs.hs`, and place it in the same directory where you will write your code. Make sure you save it with this name and capitalization. However, **do not add your code** to this file: instead, include an `import RPGdefs` directive at the top of your file.

Background

This assignment involves the components of a very simple role-playing game (RPG):

- There are two sorts of adventurers/heros: ***mages*** and ***warriors***.
- There are five sorts of items that heros might carry/hold: ***amulets***, ***gold coins***, ***potions***, ***shields***, and ***weapons***.

  Each shield provides an amount of defensive protection, represented by a numeric value. Likewise, each weapon provides an amount of offensive power, represented by a numeric value.
- Each warrior can hold an arbitrary number of items (represented by a list).

  In contrast, a mage always holds exactly one item: he may exchange one item for another, but at any instant he has one item in his possession.
- Every mage has a base offensive-power level (represented by an integer).

  Some mages (but not all) also have a special ability:

  – A mage with the **Health** ability can survive any attack, regardless of how powerful the attack might be.

  – A mage with the **Wealth** ability can purchase any item with a single coin, regardless of the item's purchase price.
- Each warrior has a subidentity, representing her preferred fighting style: ***archer***, ***swordster***, or ***brute***.

  A warrior's base offensive-power level is given by her fighting style, as shown in the following table:

Fighting Style	Base Offense

Fighting Style	Base Offense
Archer	2
Swordster	3
Brute	5

- Heroes may purchase items by using their gold. The following table gives the cost (in terms of gold coins) for the various items:

Item	Cost to Buy
Amulet	1
Gold	1
Potion	3
Level-n Shield	$2n$
Level-n Weapon	$2n + 1$

So, for example, a level-4 shield costs a hero eight gold coins to buy. A mage with the **Wealth** ability can purchase anything for one gold, regardless of the item's stated cost.

The file `RPGdefs.hs` contains Haskell datatype definitions to represent these game elements. Take a look at the file, and you'll notice two important things about these definitions:

1. None of the new types belong to the `Eq` class.

   As a result, you won't be able to use `==` or `/=` to compare elements of these types. Instead, you will need to use Haskell's pattern-matching facilities.

2. There is no limit to the number of items that a warrior can hold: for example, `Warrior Brute [Gold, Amulet, Shield 2, Weapon 1, Gold, Shield 2, Shield 4, Gold]` represents a brute who is carrying three gold pieces, one amulet, three separate shields (two at level 2 and one at level 4), and one level-1 weapon. (The order in which the items appear in the list is unimportant.)

Your Problems

For many of the functions, you will need to manipulate lists in various ways: I found list comprehensions particularly helpful and concise. You could also use recursion (a bit more cumbersome) or `map`/`filter`.

1. Define the following Haskell names/variables (they will be useful for testing out subsequent functions):

   a. A variable `mage1` of type `Hero` whose value represents a mage with the **Health** ability, who currently possesses a potion and has a base offensive power level of 5

   b. A variable `mage2` of type `Hero` whose value represents a mage with the **Wealth** ability, who currently possesses gold, and has a base offensive power level of 2

   c. A variable `mage3` of type `Hero` whose value represents a mage with the **Wealth** ability, who currently possesses a level-6 weapon, and has a base offensive power level of 4

   d. A variable `mage4` of type `Hero` whose value represents a mage with no special abilities, who currently possesses a level-5 shield, and has a base offensive power level of 7

   e. A variable `fighter1` of type `Hero` whose value represents an archer who is currently holding no items

f. A variable `fighter2` of type `Hero` whose value represents a brute who's currently holding the following: two gold pieces, an amulet, and two weapons (one at level 6 and one at level 2)

g. A variable `fighter3` of type `Hero` whose value represents an archer who is currently holding the following items: three potions, two amulets, a gold piece, and a level-3 shield

h. A variable `fighter4` of type `Hero` whose value represents a swordster who is currently holding the following items: two shields (one at level 4 and one at level 3), a level-2 weapon, a potion, two amulets, and three gold pieces

2. Write a Haskell function

```
cost :: Item -> Integer
```

such that `cost item` returns the cost (in gold) of `item`.

Examples:

- `cost Potion` evaluates to `3`
- `cost (Shield 5)` evaluates to `10`
- `cost (Weapon 5)` evaluates to `11`

3. Write a Haskell function

```
hasPotion :: Hero -> Bool
```

such that `hasPotion hero` determines whether `hero` possesses at least one potion.

- `hasPotion mage1` evaluates to `True`
- `hasPotion mage2` evaluates to `False`
- `hasPotion fighter2` evaluates to `False`
- `hasPotion fighter3` evaluates to `True`

4. Write a Haskell function

```
canBuy :: Hero -> Item -> Bool
```

such that `canBuy hero item` determines whether `hero` has sufficient wealth to purchase `item`.

Examples:

- `canBuy mage2 (Shield 2)` evaluates to `True`
- `canBuy mage3 (Shield 2)` evaluates to `False`
- `canBuy fighter2 Potion` evaluates to `False`
- `canBuy fighter4 Potion` evaluates to `True`

5. Write a Haskell function

```
boostShields :: Integer -> Hero -> Hero
```

such that `boostShields k hero` adds `k` to the level of each shield that `hero` possesses.

- `boostShields 10 mage1` evaluates to `Mage 5 Health Potion`
- `boostShields 10 mage4` evaluates to `Mage 7 None (Shield 15)`
- `boostShields 10 fighter4` evaluates to `Warrior Swordster [Shield 14,Shield 13,Weapon 2,Potion,Amulet,Amulet,Gold,Gold,Gold]`

6. A hero's total offensive power is the sum of her base offensive power and the level of all of her weapons. (Note that shields are not weapons.)

   Write a Haskell function

   ```
   heroPower :: Hero -> Integer
   ```

   such that `heroPower hero` calculates the total offensive power for `hero`.

   Examples:

   - `heroPower mage3` evaluates to `10`
   - `heroPower mage4` evaluates to `7`
   - `heroPower fighter1` evaluates to `2`
   - `heroPower fighter2` evaluates to `13`
   - `heroPower fighter3` evaluates to `2`

7. A hero can survive an attack of level n (where n is an integer) provided that their total defensive protection (i.e., the sum of the levels of all of their shields) is n or higher. In addition, a mage with the **Health** ability will survive attacks of any level.

   Write a Haskell function

   ```
   canSurvive :: Hero -> Integer -> Bool
   ```

   such that `canSurvive hero force` determines whether `hero` could survive an attach of level `force`.

   Examples:

   - `canSurvive mage1 4` evaluates to `True` (Health ability)
   - `canSurvive mage3 4` evaluates to `False` (total defense is 0)
   - `canSurvive mage4 4` evaluates to `True` (total defense is 5)
   - `canSurvive fighter2 4` evaluates to `False` (total defense is 0)
   - `canSurvive fighter4 4` evaluates to `True` (total defense is 7)

8. Write a Haskell function

   ```
   bestShield :: Hero -> Integer
   ```

   such that `bestshield hero` calculates the strength of the strongest shield in `hero`'s possession. If `hero` does not possess a shield, the function returns `0`.

   Examples:

   - `bestShield mage1` evaluates to `0`
   - `bestShield mage4` evaluates to `5`
   - `bestShield fighter2` evaluates to `0`
   - `bestShield fighter4` evaluates to `4`

What to Submit

You should submit a single Haskell file that meets the following requirements:

- There is a comment with your name and `syr.edu` email address at the top of the file.
- Functions (and the names/variables from Problem 1) should be given the stated names, have correct type signatures, and appear in the order listed in this task writeup.

- For each of the requested functions, you should provide:
  - A brief comment that describes the purpose of the function (this can be taken from the task description itself)
  - The function definition, including the type signature
  - Specific test cases that can be used to verify the correctness of your function (these tests should be designed before and as you write your code, not as an afterthought)

How it Will be Graded

Programming tasks are graded using the MRUN rubric found in the syllabus:

- The code meets the submission requirements listed above. To earn an *M*, you must have working solutions for at least 6 of the functions.
  - The code meets the submission requirements listed above. To earn an *R*, you must have working solutions for at least half of the functions.
- Your code is reasonably concise and works correctly on test data that we prepare separately. ***Use wise pattern matching to reduce the number of equations required for your functions.***