Haskell: checking the vowels in a String

Asked 2 years, 2 months ago Active 2 years, 1 month ago Viewed 1k times

2

I'm new to Haskell. I wanted to write a code, which checks if there is any vowel in a String. I came up with this solution:

```
n :: Int
n = 0

vowel :: String -> Bool
vowel a
    | check n = a !! n
    | check == 'a' || 'e' || 'i' || 'o' || 'u' || 'y'   = True
    | check /= 'a' || 'e' || 'i' || 'o' || 'u' || 'y'   = check(n+1)
    | otherwise = False
```

So basically, I want to use recursion to check letter by letter if there is any vowel. I just got the error messages like:

```
Couldn't match expected type `Bool' with actual type `Char'
   * In the second argument of `(||)', namely 'y'
     In the second argument of `(||)', namely 'u' || 'y'
     In the second argument of `(||)', namely 'o' || 'u' || 'y'
  |
14 |                  | check /= 'a' || 'e' || 'i' || 'o' || 'u' || 'y'   = check(n+1)
  |
```

Where is the problem?

string haskell

edited Sep 22 '17 at 12:26
Willem Van Onsem
206k 20 189 289

asked Sep 22 '17 at 12:25
Karolina
31 3

This approach is much more complex than needed, and the syntax is quite wrong. Start by coding a `isVowel :: Char -> Bool` function. Forget about the `n` index and `!!` : they lead to bad code. Instead, proceed by structural recursion on the string, exploiting pattern matching on `[]` and `(x:xs)` . – chi Sep 22 '17 at 12:41

One-line solution: `vowel = any (`elem` "aeiouy")` . (Yes, that's all!) – leftaroundabout Sep 22 '17 at 13:00

4 Answers

11

Problems with the current approach

This piece of code has a lot of problems:

- you define a constant `n = 0`, and somehow expect this to be a default value in `check`; that is not the case;
- you first write `check n = a !! n`, and then use `check == ...`, a variable has only one type;
- In Haskell `=` is *not* an *assignment*, it is a *declaration*. Once declared you can not change the value anymore;
- You write `check == 'a' || 'e' || 'i' || 'o' || 'u' || 'y'`, but `||` binds less thight than `==`, so you wrote `(check == 'a') || 'e' || 'i' || 'o' || 'u' || 'y'`, since a character `'e'` is not a boolean (and in Haskell there is no *truthiness*), you can not use `'e'` as an operand of `||`;
- there is no index checking so even if the above problems would be resolved; `n` would eventually grow that large, that you will get an *index out of range* exception;
- you use `!!` which is not per se wrong, but it is considered inefficient (*O(n)*) and it is not a total function so *unsafe*.

Using `any`

Let us construct a function that satisfies the needs. If I understand the requirements correctly, you need to check if there is at least one element a vowel.

A character `c` is a vowel if it is `'a'`, `'e'`, `'i'`, `'o'`, `'u'`, or `'y'`. So we can write a check:

```
isVowel :: Char -> Bool
isVowel c = c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' || c == 'y'
```

But this is rather inelegant. Since a `String` is however a list of `Char`s, we can use the `elem :: Eq a => a -> [a]` function, so we can write:

```
isVowel :: Char -> Bool
isVowel c = elem c "aeiouy"
```

Now we only have to check if there is any character `c` of the string `s` that is a vowel, so we can write:

```
vowel :: String -> Bool
vowel s = any (\c -> isVowel c) s
```

We can further improve the question. It is useless to write a lambda expression of the form `\x -> f x`. Instead we can simply write `f`. So we can write:

```
vowel :: String -> Bool
vowel s = any isVowel s
vowel :: String -> Bool
vowel = any isVowel
```

finally we can make the `isVowel` function pointfree, by using `flip` :

```
isVowel :: Char -> Bool
isVowel = flip elem "aeiouy"
```

this results into:

```
isVowel :: Char -> Bool
isVowel = flip elem "aeiouy"

vowel :: String -> Bool
vowel = any isVowel
```

We can then test it. For instance:

```
Prelude> vowel "foobar"
True
Prelude> vowel "qx"
False
Prelude> vowel "bl"
False
Prelude> vowel "bla"
True
```

Using recursion

The `any :: (a -> Bool) -> [a] -> Bool` function is a higher order function: it takes a function as input which can make it hard to understand. We can write our own specialed `any` function for the `vowel` function. We are here working with a list, and usually when doing list processing there are at least two patterns we have to consider: the empty list `[]` and the non-empty list `(x:xs)` . Since `any` is rather simple this will be sufficient.

If we process the empty list, we know that there is no vowel in the string, so we can write:

```
vowel [] = False
```

in case the list is non-empty `(x:xs)` , it has a *head* (first element) `x` and a *tail* (remaining elements) `xs` . A string contains a vowel if either the first element is a vowel, or any of the remaining elements is a vowel. So we can write:

```
vowel (x:xs) = isVowel x || vowel xs
```

if we put these together (together with the `isVowel` function), we obtain:

```
isVowel :: Char -> Bool
isVowel c = elem c "aeiouy"

vowel [] = False
```

Willem Van Onsem
206k 20 189 289

3

Your code will need a bit more of attention. Yet, this specific error is quite straightforward. The operator `||` expects two `Bool` values. Values such as `'a'` , `'b'` etc. are of type `Char` , however, so you cannot use `||` here.

What you need is a series of booleans to use `||` . The first value, `check == 'a'` , is a `Bool` in theory, because the operator `==` can take two values (of some types, including `Char`) and return `Bool` . What you want, then, is to create a series of comparisons to get many booleans. So, you have to use the `==` operator to compare `check` to each vowel:

```
 | check == 'a' || check ==  'e' || check ==  'i' || check ==  'o' || check ==  'u' ||
check ==  'y'   = True
```

This is a common misconception when starting to code. In English, one can say

> if `check` is equal to `'a'` , or `'e'` , or `'i'` ...

and we try to translate it directly into code. In Haskell, however, the "or" operator (`||`) cannot figure out what what this would mean. You have make every condition explicit:

> if `check` is equal to `'a'` , or `check` is equal to `'e'` , or `check` is equal to `'i'` ...

Will your code work once you solve this problem? No. There are other problems. For example, `check` is not defined anywhere. Nonetheless, solve one problem per time and move to the next, eventually you'll get what you look for :)

edited Sep 22 '17 at 12:54

answered Sep 22 '17 at 12:43
brandizzi
21.4k 5 83 135

1

Problem 0

You've defined `n = 0` . I suspect you meant to use that as a kind of running variable that's increased in every recursion step. That's not possible in Haskell – if you define `n = 0` at the top-level, then `n` will *always* be `0` .

To have such a “running variable”, you need to make it the argument of a recursive-loop function. You seem to have attempted something like that with `check` and its `n` argument (this is then a completely different variable, which *shadows* the global `n` – shadowing can easily lead to confusion, avoid it). The standard name for such a local function is `go` , and it would be used like

```
vowel :: String -> Bool
vowel a = go 0
 where go n = case a !! n of
          ...  ->  False
          ...  ->  True
```

You use list-indexing to loop over all the elements of a list. That's clumsy and unsafe (giving an obscure error when you index outside the list)†; the preferred way to do something like this is either to use the standard folding operations (see below), or to write your recursion *on the list directly* via pattern-matching, rather than an integer indexing variable. So we can trash that `n` argument again and recurse on the list instead:

```
vowel :: String -> Bool
vowel a = go a
 where go (x:xs) = case x of
          ...  ->  False
          ...  ->  True
          ...  -> ... go xs ...
```

Note that this will automatically fail at the end of the list, because then the pattern `x:xs` can't be matched anymore. Easy and clear, just add a special clause:

```
 where go (x:xs) = case x of
          ...  ->  False
          ...  ->  True
          ...  -> ... go xs ...
       go [] = ...
```

Actually, at this point, `go` is just a local pre-definition of `vocal` , so you might as well pull the recursion to the top-level:

```
vowel :: String -> Bool
vowel [] = ...
vowel (x:xs) = ... vowel xs ...
```

Problem 2

You've tried to declare `check` in a guard-list. That's not supported – see above, `where` should be used for local definitions, alternatively `let` ‡.

Problem 3

You've tried to write multiple equality-options in a single chain of `||` . This isn't possible because each comparison takes a (in this case) number and gives a `Bool` result, so you can't just chain them. What you can do is write out `x == 'a' || x == 'e' || ...` , but that's obviously awkward. A much neater option, as suggested by Willem, is to write ``x `elem` "aeiouy"`` instead: that will just the right thing.

If you haven't come across those backticks ` `` ` yet: they take a two-argument function like `elem` and put it into *infix mode*. This is similar to the infix `+` in `n + 1` .

The nice thing about infixes is that you can make a section of them: to define the function that takes a character and compares it with every vowel, you could write

```
isVowel c = elem c "aeiouy"
```

but you could also just write

Note that I've completely eliminated the variable, through eta reduction. This is called point-free style.

Where this comes in handy: it's not really necessary to define `isVowel` as a named function for your application – you only use it for one purpose anyway, namely compare against all chars in the string. The most common way to apply an operation to all elements of a container and combine the results¶ is `foldr`. You can use it like

```
vowel = foldr ((||) . (`elem` "aeiouy")) False
```

That's a bit cryptic; fortunately this combination of logical-or and fold has a standard name: `any`. With it, you can everything as concise as

```
vowel = any (`elem` "aeiouy")
```

†Indexing into a list is also very inefficient: `!!` needs to traverse all the elements up to the requested one. This gives your attempt a complexity of $O(n^2)$, when $O(n)$ would be completely sufficient.

‡Incidentally, it *is* possible to define something in a single guard, but only by abusing the pattern guard feature.

¶To just gather the results again in a container, you can use the simpler `fmap` instead.

edited Sep 22 '17 at 13:15 answered Sep 22 '17 at 12:57
leftaroundabout
87.5k 3 126 253

0

A simple solution using higher-order functions, for intermediate Haskellers who might find this question:

```
checkVowels :: String -> String
checkVowels = any (`elem` "aeiouyw")
```

answered Sep 23 '17 at 2:43
user8174234