Let binding & Case Expression in Haskell

Principles of Programming Languages

S6CSE, Department of CSE, Amritapuri

Let binding

- Very similar to where bindings are let bindings.
- where bindings are a syntactic construct that let you bind to variables at the end of a function and the whole function can see them, including all the guards.
- Let bindings let you bind to variables anywhere and are expressions themselves, but are very local, so they don't span across guards.
- Just like any construct in Haskell that is used to bind values to names, let bindings can be used for pattern matching.

Let binding

- Consider a function that gives us a cylinder's surface area based on its height and radius:-

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

- The form is let <bindings> in <expression>. The names that you define in the let part are accessible to the expression after the in part.
- Notice that the names are also aligned in a single column.

Difference between where and let

- let puts the bindings first and the expression that uses them later whereas where is the other way around.
- The difference is that let bindings are expressions themselves. where bindings are just syntactic constructs.
- let bindings are expressions and are fairly local in their scope, they can't be used across guards.

```
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

Let binding

- They can also be used to introduce functions in a local scope:

```
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

- We could use a let in binding in a predicate and the names defined would only be visible to that predicate. The in part can also be omitted when defining functions and constants directly in GHCi. If we do that, then the names will be visible throughout the entire interactive session.

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: `boot'
```

Case Expression

- Many imperative languages have Switch case syntax: we take a variable and execute blocks of code for specific values of that variable.
- Haskell takes this concept and generalizes it: case constructs are expressions, much like if expressions and let bindings. And we can do pattern matching in addition to evaluating expressions based on specific values of a variable.

Case Expression

- Syntax :-

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ....
```

- The expression is matched against the patterns.
  - The pattern matching action is what we expect: the first pattern that matches the expression is used. If we fall through the whole case expression and no suitable pattern is found, a runtime error occurs.

The guards cannot appear inside case expressions, they have to take scope over them.

Case Expression

- To evaluate a case expression, the expression between “case” and “of” is first evaluated, then Haskell will run through all the patterns we have given it on the left of the -> symbols and try to pattern-match the value with them.
- If it finds a match, it returns the corresponding expression to the right of the -> symbol.

```
message :: String -> String
message name =
  case name of
    "Dave" -> "I can't do that."
    "Sam"  -> "Play it again."
    _      -> "Hello."
```

Case Expression

underscore (_) pattern

- This pattern matches everything in Haskell, and it's included to make sure any time our function is called in the future with something we didn't anticipate, it will still work.
- The order matters

- In this case, even if name is "Dave", the code will never get that far, because the underscore matches on everything, and it's first in the list

```
message :: String -> String
message name =
  case name of
    _      -> "Hello."
    "Dave" -> "I can't do that."
    "Sam"  -> "Play it again."
```

Case Expression

Consider

```
f 0 = 18
f 1 = 15
f 2 = 12
f x = 12 - x
```

- It is equivalent to - and, indeed, syntactic sugar* for:

```
f x =
    case x of
        0 -> 18
        1 -> 15
        2 -> 12
        _ -> 12 - x
```

**syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express.*

Case Expression

case expressions can be embedded anywhere another expression would fit

```haskell
data Colour = Black | White | RGB Int Int Int

describeBlackOrWhite :: Colour -> String
describeBlackOrWhite c =
  "This colour is"
  ++ case c of
       Black           -> " black"
       White           -> " white"
       RGB 0 0 0       -> " black"
       RGB 255 255 255 -> " white"
       _               -> "... uh... something else"
  ++ ", yeah?"
```

Writing describeBlackOrWhite this way makes let/where unnecessary

Case Expression

- Another Example

```
data Pet = Cat | Dog | Fish | Parrot String

hello :: Pet -> String
hello x =
 case x of
 Cat -> "meeow"
 Dog -> "woof"
 Fish -> "bubble"
 Parrot name -> "pretty " ++ name
```

- We can declare custom types for data in Haskell using the data keyword.
- This is called an algebraic data type because | is like an "or", or algebraic "sum" operation for combining elements of the type while separating them with a space is akin to "and" or a "product" operation.

Case Expression

Case expressions are just a way to specify actual function values, i.e., what should get computed assuming the guards / presuppositions are satisfied.

The whole case expression is a single expression, so it must result in a value of a single type.

```
ghci 119> let {lessThanTwo :: (Integral a) => a -> String;
            lessThanTwo x
               | x < 2 = case x of {
               0 -> "zero";
               1 -> "one";
               x -> "negative number"}
               | otherwise = "two or more"}

ghci 120> lessThanTwo 0
"zero"

ghci 121> lessThanTwo 1
"one"

ghci 122> lessThanTwo (-5)
"negative number"

ghci 123> lessThanTwo 5
"two or more"
```

Case Expression

Another Example :- Suppose following are the points assigned to students based on their grade.

Grade	Points
1	10
2	9
3	8
4	4
5	3
6	2
7	1
8	0

```
getPoints :: Int -> Int
getPoints grade = case grade of
    1 -> 10
    2 -> 9
    3 -> 8
    4 -> 4
    5 -> 3
    6 -> 2
    7 -> 1
    8 -> 0
    _ -> -1
```

Next - List